University of Stuttgart
Institute of Industrial Automation and
Software Engineering

An Auto-Encoder implementation for software quality attributes estimation using complexity metrics

Research Project FA-3569

Submitted at the University of Stuttgart by
Nikita Majalikar

Electrical Engineering

Examiner:	Prof. Dr.-Ing. Michael Weyrich
Supervisor:	Golsa Ghasemi

27.11.2023

Table of Contents

Table Of Figures

Table of Tables

Table of Abbreviations

DNN	**D**eep **N**eural **N**etwork
TBF	**T**ime **B**etween **F**ailure
PCA	**P**rincipal **C**omponent **A**nalysis
LOC	**L**ines **O**f **C**ode
DIT	**D**epth **O**f **i**nheritance
VAE	**V**ariational **A**utoencoder
CNN	**C**onvolutional **N**eural **N**etwork
FNN	**F**eedforward **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory **N**etwork
GTU	**G**ated **R**ecurrent **U**nit
RPA	**R**obotic **P**rocess **A**utomation
ATDD	**A**cceptance **T**est **D**riven **D**evelopment
MSE	**M**ean **S**quared **E**rror
MAE	**M**ean **A**bsolute **E**rror

Glossary

Autoencoder	A type of artificial neural network used for unsupervised learning to encode input data into a lower-dimensional representation
Deep Neural Network	A neural network with multiple hidden layers, allowing it to learn intricate features and patterns from complex data
Bijective Mapping	A mapping between two sets that is both injective (one-to-one) and surjective (onto), ensuring a unique correspondence between elements
Mean Squared Error	A measure of the average squared difference between predicted and actual values, commonly used as a loss function in regression tasks
Mean Absolute Error	A measure of the average absolute difference between predicted and actual values, providing a more interpretable error metric
Feedforward Neural Network	A type of neural network architecture where information moves in one direction, from input to output
Dimensionality Reduction	The process of reducing the number of input features while preserving essential information

Abstract

Analyzing software complexity and quality is of paramount importance in contemporary software engineering practices. The intricate nature of software systems necessitates advanced techniques for accurate estimation and prediction of quality attributes. This thesis addresses the challenge by introducing an innovative approach that combines artificial neural networks, specifically autoencoders, with complexity metrics.

The problem at hand revolves around the difficulty in precisely estimating software quality attributes, including complexity and consequences. Conventional methods often fall short in capturing the intricacies of high-dimensional, non-linear software requirement vectors. To overcome this, the proposed approach leverages the capabilities of autoencoders to perform dimensionality reduction and feature learning. Autoencoders, adept at capturing both linear and non-linear relationships, provide a robust framework for obtaining a compressed representation of software requirements.

The implementation involves training the autoencoder on input vectors representing software requirements, resulting in a lower-dimensional representation. Subsequently, separate Deep Neural Networks (DNNs) are developed for complexity and consequence predictions based on these compressed requirement vectors. The complexity DNN estimates fundamental software complexity metrics, such as lines of code and cyclomatic complexity. Simultaneously, the consequence DNN predicts metrics related to software behavior, including the number of calls, time between failures (TBF), and reliability.

In summary, this thesis makes a significant contribution to the field of software engineering by introducing an innovative methodology for estimating software quality attributes. The fusion of autoencoders and DNNs, complemented by the integration of complexity metrics, yields improved accuracy in predictions. This advancement not only enhances software quality assessment but also fosters the development of high-quality software solutions.

Key Words: *Software quality attributes, Autoencoder, Deep Neural Networks, Complexity metrics, Complexity*

1 Introduction

In the dynamic landscape of digital transformation, software serves as the invisible architect, intricately shaping the contours of our interconnected world. From healthcare and finance to transportation and entertainment, the software underpinning modern systems influences our world to a profound extent. Within this digital realm, the quality of software becomes a non-negotiable requirement. A minor software glitch can have catastrophic consequences, leading to substantial financial losses, damage to an organization's reputation, and, in some cases, endangering lives. In the ever-evolving field of software engineering, maintaining the quality of software poses a significant and ongoing challenge. The question that arises is this: How do we guarantee the quality of software when it operates within an intricate, ever-evolving environment?

Stable and competetive product
Saves money and time
Safety
QUALITY ASSURANCE
Meets clients demands
Reputation
New view on your project

Figure 1: Importance of quality assurance [1]

Software quality is a multifaceted concept, encompassing attributes such as reliability, maintainability, performance, and security. Estimating and predicting these attributes is pivotal in software engineering, as it allows developers and stakeholders to make informed decisions, allocate resources efficiently, and address issues proactively. Among these quality attributes, one of the most influential factors is the complexity of software systems. As shown in Figure 1, these metrics are essential tools for quality assessment, offering a quantitative lens through which developers and quality assurance professionals can evaluate the robustness and reliability of software. By identifying high-risk areas through metrics like Cyclomatic Complexity, development teams can strategically allocate resources, focusing testing and debugging efforts where they are most needed. Furthermore, complexity metrics serve as valuable predictors of maintainability, helping teams anticipate challenges in code understanding and modification over the software's lifecycle. During code reviews, these metrics provide objective criteria for evaluating clarity, guiding developers in refactoring efforts to enhance code readability and maintainability. The understanding of software complexity also aids in performance optimization by pinpointing potential bottlenecks. Ultimately, these metrics contribute to risk mitigation strategies, offering early insights into potential challenges and enabling proactive measures to ensure the development of high-quality software systems. In essence, complexity metrics are indispensable instruments that empower software engineers to navigate the intricate landscape of software development, optimize resources, and deliver resilient and efficient software solutions.

However, quantifying and estimating software quality attributes, particularly in the context of high-dimensional data such as software complexity metrics, remains a persistent challenge. Traditional methods and tools, while valuable, often struggle to comprehensively address the complex nature of software systems [2]. The era of big data has ushered in a deluge of information, making manual analysis and prediction an impractical endeavor.

This is where our research takes center stage. In this thesis, we present a pioneering approach to estimating software quality attributes. Our proposal leverages autoencoders, a subset of artificial neural networks known for their effectiveness in dimensionality reduction and feature learning. Autoencoders, when applied to the realm of software engineering, offer the promise of accurate and reliable estimation of software quality attributes. By delving into the intricate realm of complexity metrics and other software attributes, our research explores the transformative potential of autoencoders.

2 Literature Survey

This section presents a comprehensive review of the traditional methods employed for predicting software quality metrics before delving into contemporary approaches such as PCA and autoencoders. It also thoroughly investigates the various aspects such as complexity metrics and their diverse types. Additionally, the literature survey thoroughly investigates in-depth analysis of different complexity metrics utilized in software engineering, emphasizing their significance and applications.

2.1 Traditional Methods for Predicting Software Quality Metrics

In the realm of software engineering, traditional methods have long been employed to predict software quality metrics. Some notable approaches include:

2.1.1 Regression Models

Regression models, including linear regression and polynomial regression, have been widely used to establish relationships between various software metrics and quality attributes. These models leverage statistical techniques to predict the impact of different factors on software quality.

2.1.2 Statistical Analysis

Statistical methods, such as correlation analysis and ANOVA, have been instrumental in identifying and quantifying the relationships between different software metrics and overall software quality. These analyses provide insights into the significance of individual metrics in predicting quality outcomes.

2.1.3 Threshold-Based Approaches

Setting predefined thresholds for specific software metrics has been a conventional method for predicting software quality. Projects exceeding or failing to meet these thresholds are categorized based on their expected quality outcomes.

2.2 Principal Component Analysis (PCA)

PCA is a dimensionality reduction technique that linearly transforms data into a new coordinate system to reduce its dimensionality while retaining most of the variance. It involves finding the principal components of the data, which are orthogonal linear combinations of the original features [18].

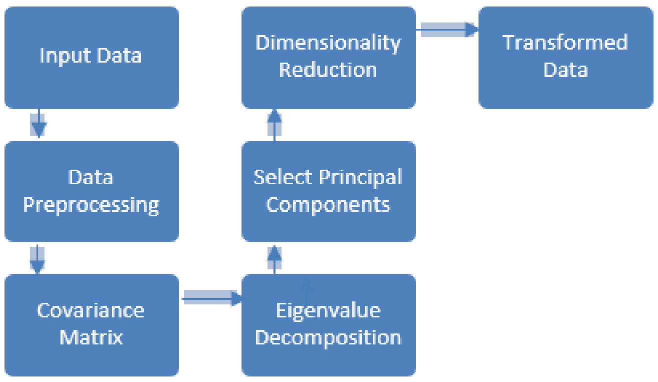


Figure 2: Overview of PCA process

Figure 2 shows the overview of PCA process to obtain the lower-dimensional representation of the data starting with the input data. The steps are mentioned below:

1. **Data Preprocessing:**
   If the features have different scales, it is important to preprocess them to have zero mean and unit variance. This ensures that each feature contributes equally to the analysis.

2. **Compute the covariance matrix:**
   Calculate the covariance matrix to understand the relationships between different features in the dataset. The covariance matrix represents how changes in one variable are associated with changes in other variables.

3. **Calculate the eigenvalues and eigenvectors:**
   Compute the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors represent the principal components, and the corresponding eigenvalues represent the amount of variance explained by each principal component. The eigenvectors are orthogonal to each other, meaning they are uncorrelated.

4. **Select the principal components:**
   Choose the desired number of principal components to retain based on the explained variance ratio or a desired level of dimensionality reduction. The principal components are ranked in descending order of the corresponding eigenvalues.

5. **Transform the data:**
   Project the original data onto the selected principal components to obtain the lower-dimensional representation of the data.

Reason for not using PCA: PCA is a powerful technique for dimensionality reduction, but it is inherently linear. It assumes that the relationships between variables are linear.

In the case of software quality attributes estimation, it's common for the relationships between requirements and attributes to be nonlinear and complex. Autoencoders, on the other hand, can capture both linear and nonlinear relationships in the data, making them a better choice when dealing with complex, high-dimensional data.

In conclusion, the exploration of Principal Component Analysis (PCA) as a dimensionality reduction technique reveals its efficacy in transforming data into a new coordinate system while retaining significant variance. The stepwise process of standardizing data, computing the covariance matrix, and deriving eigenvalues and eigenvectors elucidates the principal components, offering a valuable tool for reducing dimensionality in a linear context. However, the decision not to utilize PCA in the context of software quality attributes estimation stems from its inherent linearity assumption. Recognizing that relationships between requirements and attributes can be nonlinear and complex in software engineering, the preference leans toward autoencoders. Autoencoders exhibit the capability to capture both linear and nonlinear relationships within data, making them a more suitable choice for addressing the intricate and high-dimensional nature of software quality attribute estimation. This informed decision highlights the importance of choosing dimensionality reduction techniques that align with the inherent complexities of the specific domain under investigation.

2.3 Autoencoders

An autoencoder is a type of artificial neural network used for unsupervised learning and dimensionality reduction. It is primarily used for data compression, feature extraction, and reconstruction tasks. As shown in Figure 3, autoencoders consist of an encoder network and a decoder network, which work together to learn a compressed representation of the input data [9].

The encoder takes the input data and maps it to a lower-dimensional latent space representation, often referred to as a bottleneck or encoding. This encoding typically has fewer dimensions than the original input data, forcing the network to capture the most important features of the input while discarding some of the less significant details. The decoder network takes the compressed representation and attempts to reconstruct the original input data from it. By training the autoencoder to minimize the reconstruction error, it learns to encode and decode the data efficiently, preserving important information while eliminating noise or irrelevant features [9].

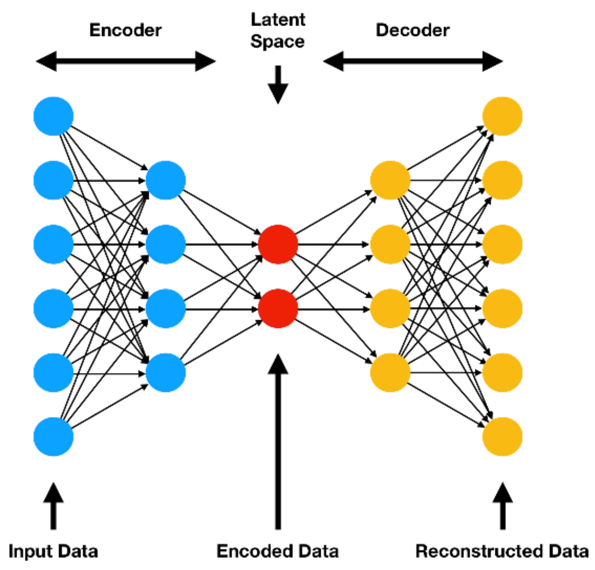


Figure 3: Structure of Autoencoder [10]

Autoencoders can be used for various applications, including dimensionality reduction, data denoising, anomaly detection, and even generative modelling. There are different types of autoencoders, each with its specific variations and objectives. All autoencoder types, however, follow the same basic idea of rebuilding the input after learning a compressed representation.

2.3.1 Different types of Auto-Encoders

The family of autoencoders encompasses various types, each designed for specific tasks and data characteristics. Here, we introduce some of the prominent types of autoencoders relevant to our quest for software quality attribute estimation:

2.3.1.1 Vanilla Auto-Encoder

Also known as a basic or traditional autoencoder, it consists of an encoder and a decoder network as shown in Figure 4. Its goal is to minimize the reconstruction error between the original input and the output produced by the decoder in order to learn a compressed version of the input data. One of the key features of the vanilla autoencoder is that the dimensionality of the latent space or bottleneck layer is usually smaller than the input and output dimensions [11]. This forces the network to learn a compressed representation, which captures the essential information while discarding some of the less significant details [9].

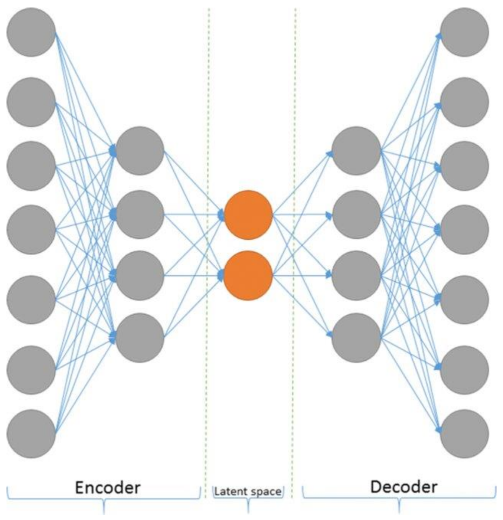


Figure 4: Structure of Vanilla Autoencoder [12]

Advantages:

- Simple and easy to implement

Disadvantages:

- Prone to overfitting and may struggle with complex data

Application:

- Data denoising, anomaly detection, and feature extraction.

2.3.1.2 Sparse Auto-Encoder

Sparse autoencoders incorporate sparsity constraints during training, encouraging the autoencoder to learn sparse representations where only a few units in the latent space are active for a given input. This promotes more efficient and meaningful representations, making it useful for feature selection and anomaly detection [9]. Figure 5 shows the structure of sparse autoencoder.

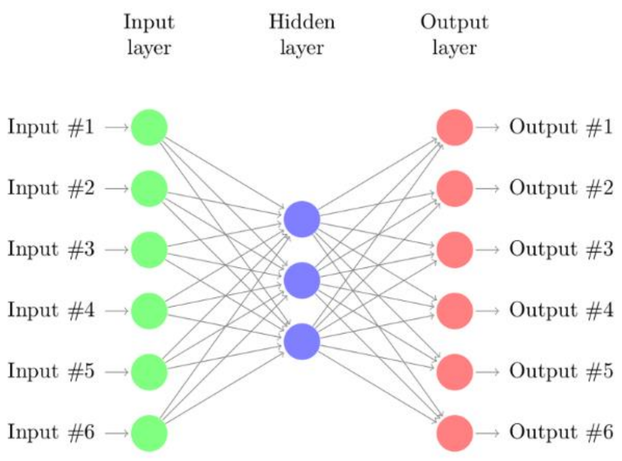


Figure 5:Structure of Sparse Autoencoder [13]

Advantages:

- Sparsity penalty prevents overfitting.
- They take the highest activation values in the hidden layer. This prevents the use all of the hidden nodes at a time.

Disadvantages:

- The individual nodes of a trained model which activate are data Dependent
- Different inputs will result in activations of different nodes through the network.

Application:

- Data denoising, anomaly detection, and feature extraction.

2.3.1.3 Denoising Auto-Encoder

Denoising autoencoders are trained to reconstruct the original input from a corrupted or noisy version of it. By exposing the autoencoder to corrupted data as shown in Figure 6 and requiring it to produce accurate reconstructions, it learns to extract meaningful features and eliminate noise. Denoising autoencoders can be effective for noise reduction and robust feature learning [9].

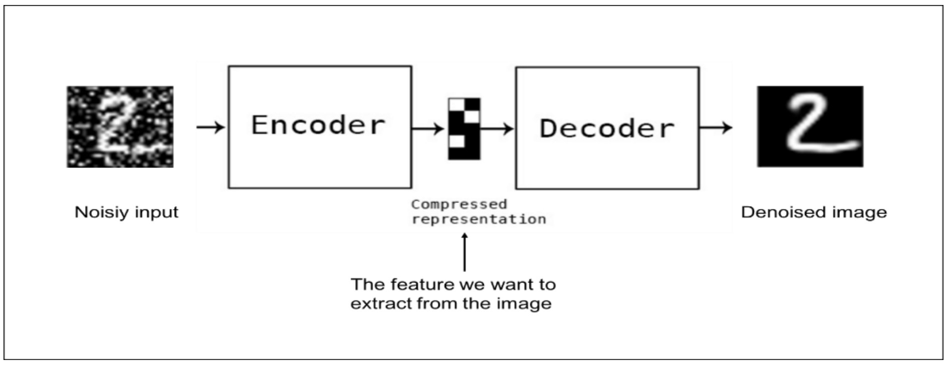

Encoder
Decoder
Noisiy input
Compressed representation
Denoised image
The feature we want to extract from the image

Figure 6: Structure of Denoising Autoencoder [14]

Advantages:

- learn robust representations that can extract meaningful information from noisy or corrupted data.
- Minimizes the loss function between the output node and the corrupted input.
- Setting up a single-thread denoising autoencoder is easy.

Disadvantages:

- To train an autoencoder to denoise data, it is necessary to perform preliminary stochastic mapping in order to corrupt the data and use as input.
- This model isn't able to develop a mapping which memorizes the training data because our input and target output are no longer the same

Application:

- image denoising, speech enhancement, and data preprocessing

2.3.1.4 Contractive Auto-Encoder

Contractive autoencoders add a regularization term to the loss function that penalizes the sensitivity of the model to small perturbations in the input space. This encourages the autoencoder to learn robust and invariant features, making it useful for tasks such as feature extraction and denoising [9]. The Structure of contractive autoencoder is shown in Figure 7.

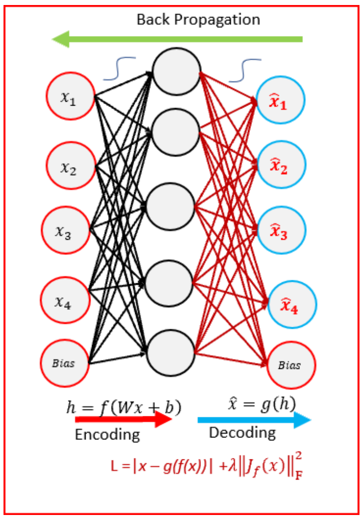


Figure 7: Structure of contractive Autoencoder [11]

Advantages:

- Contractive autoencoders learn useful feature extraction.
- This model learns an encoding in which similar inputs have similar encodings.
- Hence, we're forcing the model to learn how to contract a neighborhood of inputs into a smaller neighborhood of outputs.

Disadvantages:

- Increased computational complexity
- Difficulty in tuning hyperparameters

Application:

- data denoising, anomaly detection, or feature extraction.

2.3.1.5 Variational Autoencoder

Variational Auto-encoders (VAEs) are generative models that learn a probabilistic distribution in the latent space. They introduce a regularization technique that forces the latent space to follow a prior distribution (typically a Gaussian distribution) as shown in Figure 8. VAEs can generate new samples by sampling from the learned distribution and are useful for tasks such as data generation and interpolation [9].

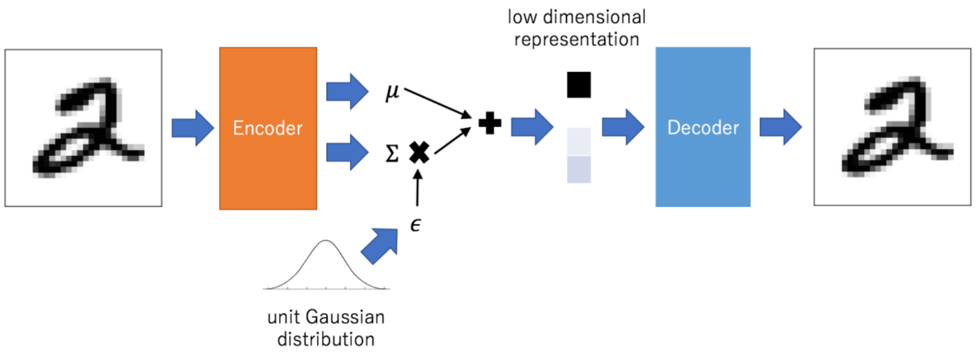


Figure 8: Structure of Variational Autoencoder [15]

Advantages:

- It gives significant control over how we want to model our latent distribution, unlike the other models.
- After training you can just sample from the distribution followed by decoding and generating new data.

Disadvantages:

- When training the model, there is a need to calculate the relationship of each parameter in the network concerning the final output loss using a technique known as backpropagation. Hence, the sampling process requires some extra attention

Application:

- image generation, text synthesis, and data augmentation.

2.3.1.6 Convolutional Autoencoder

A convolutional autoencoder is a type of autoencoder architecture that utilizes convolutional neural networks (CNNs) as its building blocks. It is specifically designed to handle input data with a grid-like structure, such as images. The structure of convolutional autoencoder is shown in Figure 9. By leveraging convolutional layers, convolutional autoencoders can capture spatial patterns and hierarchical representations effectively [16].

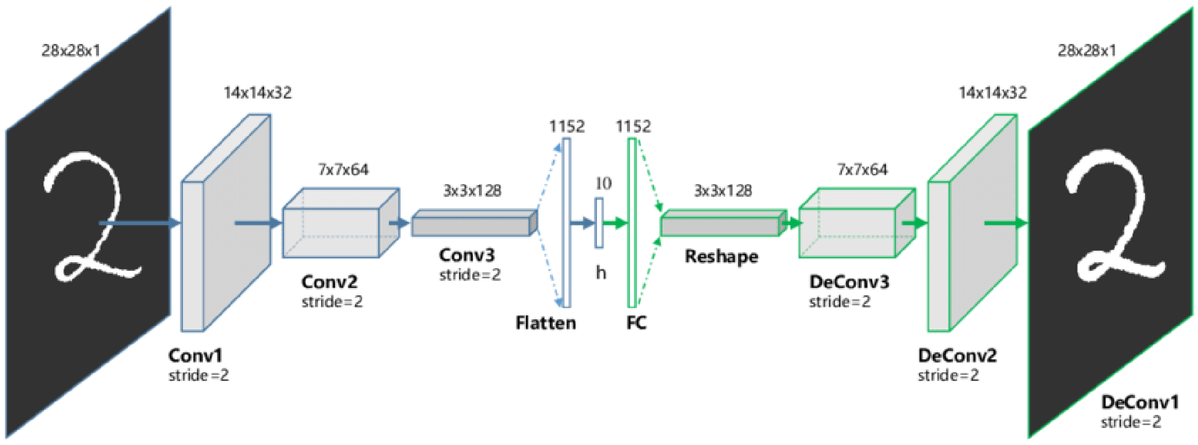


Figure 9: Structure of Convolutional Autoencoder [17]

Advantages:

- Due to their convolutional nature, they scale well to realistic-sized high-dimensional images.
- Can remove noise from pictures or reconstruct missing parts.

Disadvantages:

- The reconstruction of the input image is often blurry and of lower quality due to compression during which information is lost

Application:

- image denoising, image generation & feature extraction

In summary, autoencoders represent a powerful class of artificial neural networks extensively utilized for unsupervised learning and dimensionality reduction. Comprising an encoder network and a decoder network, they learn to compress input data into a lower-dimensional latent space, emphasizing essential features while discarding less significant details. The family of autoencoders encompasses various types, each tailored for specific tasks. Vanilla autoencoders, for instance, focus on basic encoding and decoding, while sparse autoencoders introduce sparsity constraints, promoting more efficient representations. Denoising autoencoders specialize in reconstructing original inputs from noisy versions, contributing to robust feature learning. Contractive autoencoders add regularization for robustness against perturbations, and variational autoencoders introduce probabilistic distributions in the latent space, enabling tasks like data generation. Convolutional autoencoders, incorporating convolutional neural networks, are tailored for grid-like data structures such as images, capturing spatial patterns effectively. Understanding these diverse autoencoder types lays the foundation for exploring their application in predicting software quality attributes, bridging the literature gap between traditional methods and contemporary approaches.

2.4 Complexity Metrics

The field of software development has recently faced enormous problems due to the introduction of new technologies like artificial intelligence, blockchain, and the Internet of Things. These technologies have increased software system complexity, which makes it more difficult for developers to design, create, and maintain software that meets end user demands. As a result of these technologies, the complexity of software systems has increased, making it harder for developers to design, build, and maintain software programs that fulfil the demands of end users. Metrics for software complexity give a quantitative method for assessing and controlling program complexity [2]. This metric is extremely important in software management and plays a significant influence on project success. During the development phases of software, the amount of effort required to evaluate requirements, design, code, test, and debug the system is heavily influenced by complexity as shown in Figure 10. They act as a powerful diagnostic tool, revealing the underlying intricacies that have a profound impact on a software system's reliability, maintainability, and overall performance [2].

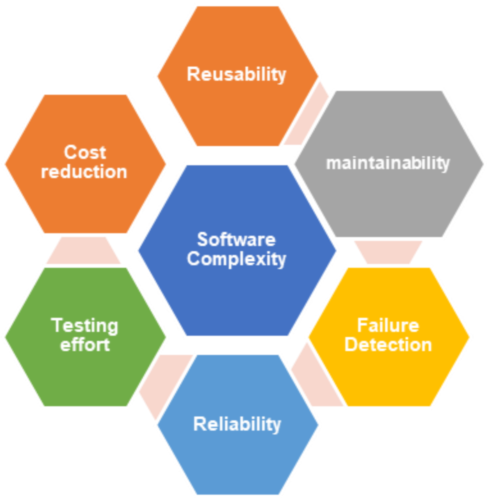


Figure 10: Significance of Software Complexity metrics

One of the primary reasons for the significance of complexity metrics is their predictive power. By analyzing software complexity, we can foresee potential trouble spots, such as areas susceptible to defects or those likely to suffer performance bottlenecks [3]. Consequently, complexity metrics empower software engineers and stakeholders to make informed decisions regarding resource allocation, risk management, and quality assurance.

2.4.1 Types Of Complexity Metrics

The world of software complexity is not monolithic; rather, it comprises various facets that necessitate diverse metrics for comprehensive measurement. Numerous complexity measures have been established over time by academics and industry professionals, each concentrating on a different facet of software systems. They are broadly classified into static and dynamic complexity metrics.

2.4.1.1 Static Complexity Metrics

Static complexity metrics are measures that are derived directly from the source code without executing the program [4].

Here, are some of the prominent types of static complexity metrics:

1) **Cyclomatic Complexity (McCabe's Complexity):** This metric quantifies the complexity of control flow within a software program. It is instrumental in identifying the number of independent paths in the program, which corresponds to the number of test cases required for thorough testing [5].
2) **Lines of Code (LOC):** Measuring the size and complexity of software based on the number of lines of code is a traditional approach. It provides insights into a software system's overall scope, but it has limitations in assessing the structural complexities within the code [4].
3) **Halstead Complexity Measures:** These metrics focus on quantifying the complexity of software through mathematical operators and operands. By analyzing the number of unique operators and operands, Halstead complexity metrics offer a distinct perspective on software complexity [6].
4) **Maintainability Index:** This metric gauge how maintainable a software system is, which is a crucial aspect of software quality. It factors in various attributes, including code size, cyclomatic complexity, and comment density [3].
5) **Depth of Inheritance Tree (DIT):** DIT quantifies the level of inheritance within a software system's class hierarchy. It helps assess the potential complexities associated with class relationships and inheritance [7].
6) **Coupling and Cohesion Metrics:** These metrics focus on the relationships between software modules. High coupling or low cohesion can indicate potential complexities related to code modularity and maintainability [3].

2.4.1.2 Dynamic Complexity Metrics

Dynamic complexity metrics are measures that are obtained by executing the program and observing its behavior at runtime [4] [8].
Here, we introduce some of the prominent types of static complexity metrics:

1) **Execution Time:**
   Measures the time taken by the program to execute a specific task or a set of tasks. If a program takes an unexpectedly long time to execute a function, it might indicate a performance issue that needs optimization [8].

2) **Memory Usage:**
   Examines the amount of memory that the program consumes during execution. High memory usage can lead to performance issues, and monitoring it helps ensure efficient resource utilization and avoid memory-related problems [5].

3) **Code Coverage:**
Code coverage measures the percentage of code that is exercised by a set of tests. Achieving high code coverage indicates that a significant portion of the code has been tested, increasing confidence in the reliability of the software [4].

4) **Number of Bugs:**
Tracks the number and severity of bugs encountered during program execution. A decreasing number of bugs over time may indicate improvements in code quality and reliability [4].

Table 1: Summary Of ComplexityMetrics

Parameter	Type	Usage	Significance
Cyclomatic Complexity	Static Complexity	Control flow complexity assessment	Identifies potential testing needs
Execution Time	Dynamic Complexity	Performance evaluation	Flags potential optimization requirements
Lines of Code	Static Complexity	Software size and complexity measurement	Offers insights into overall code scope
Memory Usage	Dynamic Complexity	Memory consumption monitoring	Ensures efficient resource utilization
Maintainability Index	Static Complexity	Assessing code maintainability	Considers code size, complexity, and comments
Depth of Inheritance Tree	Static Complexity	Quantifies class hierarchy inheritance level	Aids assessment of class relationships
Code Coverage	Dynamic Complexity	Percentage of code tested	Indicates reliability through thorough testing
Number of Bugs	Dynamic Complexity	Tracking bug count and severity	Reflects improvements in code quality

The journey into the realm of software complexity metrics is multifaceted, with each metric offering a unique lens into software quality. It is through this rich tapestry of metrics that we embark on our quest to enhance software quality attribute estimation.

2.5 Literature Gaps

Following an extensive literature survey, several critical gaps in the existing body of knowledge have come to light, warranting focused attention and investigation in the realm of software quality attribute estimation. One prominent gap pertains to the limited exploration of advanced techniques for predicting software quality attributes, particularly in the context of complexity and consequences. While traditional methods such as regression models, statistical analysis, and threshold-based approaches have been foundational, there is a discernible lack of comprehensive approaches that harness the power of artificial neural networks, such as autoencoders, for a nuanced understanding of intricate software systems.

Another notable gap is evident in the insufficient integration of complexity metrics with contemporary machine learning models. Although complexity metrics serve as indispensable tools for evaluating software intricacies, their synergy with advanced neural network architectures remains underexplored. The literature often falls short in offering a holistic framework that seamlessly incorporates both complexity metrics and artificial neural networks, hindering the development of robust and accurate predictive models.

Furthermore, there is a dearth of research that addresses the multifaceted nature of software quality attributes. While existing studies may focus on isolated aspects, such as complexity or consequences, there is a paucity of comprehensive approaches like autoencoders that concurrently consider these attributes. The interplay between complexity and consequences in influencing overall software quality remains an underexplored domain, necessitating a more integrated and holistic research approach.

Additionally, the literature review highlights a gap in the exploration of autoencoders specifically for dimensionality reduction and feature learning in the context of software requirement vectors. The potential of autoencoders to capture both linear and non-linear relationships within high-dimensional data has not been fully harnessed in the domain of software quality attribute estimation. This presents a notable research opportunity to delve into the effectiveness of autoencoders in enhancing the accuracy and efficiency of predictive models for software quality attributes.

In addressing these identified gaps, this research endeavors to contribute to the existing body of knowledge by proposing an innovative framework that amalgamates complexity metrics, artificial neural networks, and autoencoders. By doing so, it seeks to bridge these critical gaps and advance the understanding and prediction of software quality attributes in contemporary software engineering practices.

3 Conception

This chapter talks about the architecture that is considered for this research and the individual components such as autoencoder, DNN and their selection criteria. It also discusses the other possible approaches that were available but had to be backed out due to its limitations.

3.1 Architecture

This Research thesis follows a step-by-step approach, combining the power of autoencoders and neural networks to compress features and make predictions. The autoencoder, a pivotal neural network architecture, spearheads the initial stage by engaging in data compression through its encoder-decoder structure. This involves mapping input data to a lower-dimensional latent space and reconstructing it, with the objective of minimizing reconstruction error, thus rendering the autoencoder adept at both dimensionality reduction and feature extraction. Normalization precedes this step, ensuring equitable scales across diverse features to prevent dominance based on their individual scales.

Following data compression, a deep neural network (DNN) is deployed for a bijective mapping, aiming to recover complexity values such as 'Lines of Code' and 'Cyclomatic Complexity' from the compressed representation. This intricate mapping establishes the model's capacity to decode the compressed data back into its original form with minimal error. Subsequently, the neural network extends its predictive capabilities to anticipate consequences, including 'Number of Calls,' 'Time Between Failures (TBF),' and 'Reliability.' These predictions are derived using the same compressed representation obtained from the autoencoder.

Integral to the evaluation of both complexity and consequence predictions are metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics serve as vital tools in scrutinizing the accuracy of the model's predictions, providing a comprehensive assessment of its performance. This thesis navigates the intricate landscape of software engineering, showcasing the efficacy of a combined approach involving autoencoders and neural networks for feature compression and consequential predictions.

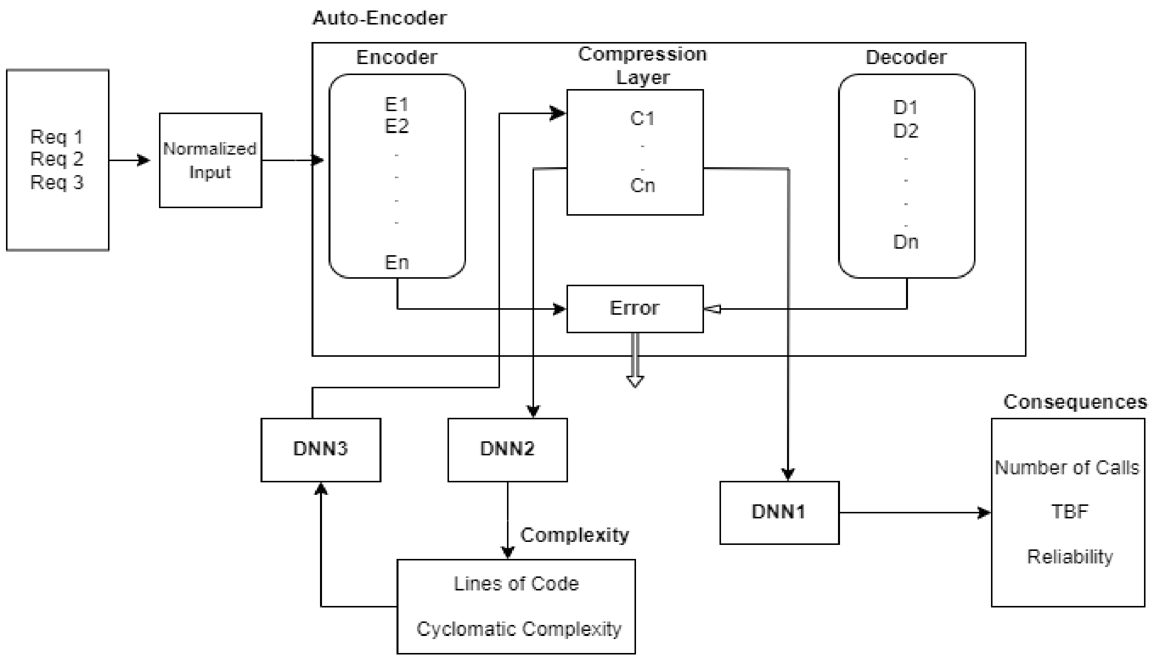


Figure 11: Architecture of proposed methodology

This architecture shown in Figure 11 is a multi-step process involving autoencoders and neural networks for feature compression and prediction. Here's a brief explanation of the concepts

1) **Autoencoder for Data Compression:**
   An autoencoder is a neural network architecture used for dimensionality reduction and feature compression. It consists of an encoder and a decoder. The encoder maps the input data into a lower-dimensional latent space representation, and the decoder reconstructs the input data from this lower-dimensional representation. The objective of an autoencoder is to minimize the reconstruction error, which makes it useful for data compression and feature extraction. However, the learned features from an autoencoder can potentially be used as inputs for classification or regression models in a subsequent step [9].

2) **Normalization:**
   Before feeding the data into the autoencoder, it's normalized it to bring all features within a similar scale. Normalization is crucial to ensure that different features do not dominate the learning process based on their scales [19].

3) **Mean Square Error (MSE):**
   The Mean Square Error between the input data and the data reconstructed by the autoencoder is calculated. This MSE serves as a measure of how well the autoencoder can capture and reproduce the input data [20].

4) **Bijective Mapping for Complexity:**
After compressing the data using the autoencoder, a DNN for bijective mapping to retrieve complexity values is set up. This mapping enables us to decode the compressed representation back into 'Lines of Code' and 'Cyclomatic Complexity'. The model learns the inverse mapping to reconstruct these features with minimal error.

5) **Predicting Consequences:**
We further extend your neural network to predict consequences, including 'Number of Calls,' 'TBF,' and 'Reliability. The model uses the same compressed representation from the autoencoder to make these predictions.

6) **Model Evaluation:**
In both complexity and consequence predictions, we evaluate the models using metrics like Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics help assess the accuracy of the model's predictions.

3.2 Choosing Autoencoder

In this context, a straightforward feedforward autoencoder architecture is employed, comprising three main layers: an input layer, an encoder layer, and a decoder layer. This architecture is designed to facilitate the encoding and subsequent decoding of input data, enabling the extraction and representation of essential features. To provide a detailed overview, the process starts with the input layer, where the original data is fed into the system. The encoder layer follows, tasked with compressing the input data into a reduced-dimensional representation, effectively capturing its crucial features. Subsequently, the decoder layer works to reconstruct the original data from this compressed representation. This simple yet effective autoencoder architecture in Figure 12 serves as a foundational component for feature learning and dimensionality reduction, contributing to the overall success of the methodology.

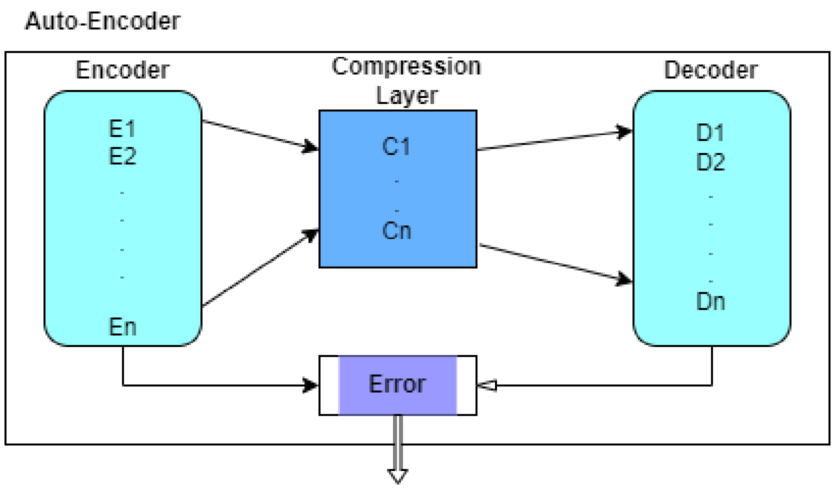


Figure 12: Architecture of Autoencoder

Input Layer: This layer represents the input data, and it has a shape of (YOUR_VECTOR_LENGTH,), which is determined by the length of input vectors.

Encoder Layer: Here, the encoder layer is defined with 64 neurons and uses the ReLU (Rectified Linear Unit) activation function. This layer is responsible for reducing the dimensionality of the input data and capturing essential features. The encoder layer reduces the input vector of length YOUR_VECTOR_LENGTH to a lower-dimensional representation with 64 dimensions.

Decoder Layer: The decoder layer has only one neuron with a sigmoid activation function. This layer aims to reconstruct the input data from the compressed representation created by the encoder. The output of the decoder layer has the same shape as the input data.

The code uses the Mean Squared Error (MSE) loss function, which is a common choice for autoencoders. MSE measures the average squared difference between the input data and the reconstructed data. The autoencoder is trained to minimize this loss, ensuring that the reconstruction is as close to the original input as possible.

This type of autoencoder is referred to as a "Denoising Autoencoder" because it aims to learn a compact representation of the input data while filtering out noise and irrelevant features. The choice of the activation functions (ReLU in the encoder and sigmoid in the decoder) and the layer sizes (64 neurons in the encoder) is a common configuration for a basic autoencoder.

The purpose of this autoencoder is to reduce the dimensionality of the input data (from YOUR_VECTOR_LENGTH to 64) and learn a compressed representation of the data. This can be useful for various purposes, such as feature extraction, data compression, or as a preprocessing step for other machine learning models. It is a general-purpose autoencoder that can be adapted for tasks such as dimensionality reduction and denoising, depending on how it is trained and used.

3.3 DNN for Complexity Prediction

In the context of complexity analysis, deep neural networks (DNNs) can be used for various purposes, depending on the specific problem and the type of data we have. The choice of the DNN architecture depends on whether we're dealing with regression or classification tasks and the nature of the data we're working with. Here are some common types of DNN architectures used for complexity analysis:

1) **Feedforward Neural Networks (FNNs):** FNNs are one of the simplest forms of DNNs. They consist of input layers, hidden layers, and output layers. They can be used for regression tasks, such as predicting numerical values like lines of

code or cyclomatic complexity, by having a single output neuron in the output layer [21].

2) **Convolutional Neural Networks (CNNs):** CNNs are commonly used for analyzing complex data like images or sequences, but they can also be applied to complexity analysis. They are useful when the data has a grid-like structure, and they can automatically learn hierarchical features. In this case, the input data would need to be structured in a way that makes sense for a CNN [21].

3) **Recurrent Neural Networks (RNNs):** RNNs are designed for sequential data and are useful when dealing with time series data or data with a sequential nature. For example, if you are analyzing complex data over time, RNNs can be suitable [22].

4) **Long Short-Term Memory Networks (LSTMs):** LSTMs are a type of RNN that is particularly effective in handling sequences with long-range dependencies. They can be used when the complex data exhibits complex temporal patterns [22].

5) **Gated Recurrent Units (GRUs):** GRUs are another type of RNN and can be used in situations similar to LSTMs but with a simplified structure [22].

6) **Custom Architectures:** Depending on the specific nature of your complexity analysis problem, we may design custom DNN architectures. This could involve combining various layers and units to best suit our data.

3.3.1 Bijective Mapping DNN

The bijective mapping model is a custom DNN architecture that takes the compressed data from the compression layer as input and predicts the actual complexity values (Lines of Code and Cyclomatic Complexity) as output. This mapping is designed to be bijective, meaning it can map the compressed data to complexity values and vice versa.

The DNN architecture used for this specific task is not a standard classification or regression network. It's designed for mapping between two representations while ensuring a bijective relationship. This is a specialized use case based on our requirements and data structure.

This approach allows us to efficiently encode and decode complex data while preserving a reversible mapping between the compressed representation and the actual complexity values, making it useful for data reconstruction and other tasks.

3.4 DNN for Consequence Prediction

The term ‘consequence’ likely refers to the outcomes or results associated with software engineering processes or decisions. It might encompass various aspects such as the impact of changes, reliability of the software, and the consequences of different design choices. Consequences can be evaluated in terms of both positive and negative effects on software quality, performance, and overall system behavior. Here the attributes considered for consequences are mentioned below:

1) Number of Calls: The frequency of function or method calls within the software, which can impact performance and resource utilization.
2) Time Between Failures (TBF): The duration between occurrences of software failures, indicating the system's reliability and stability.
3) Reliability: The overall dependability and trustworthiness of the software, reflecting its ability to perform as expected under various conditions.

Consequence prediction means understanding and forecasting the outcomes or results associated with certain software attributes. A Deep Neural Network (DNN) is used for predicting consequences. Specifically, the DNN architecture used in this context is a feedforward neural network with one hidden layer. Here's more detail on the DNN architecture for predicting consequences:
Input Layer: The input layer of the DNN has 64 units. This input size corresponds to the compressed representation obtained from the autoencoder.

Hidden Layer: The DNN contains one hidden layer with a Dense layer having 3 output neurons. The activation function for this hidden layer is 'linear,' meaning it performs a linear transformation. This hidden layer captures the relationships and patterns in the compressed representation and maps them to the 3 consequence attributes: 'Number of Calls', 'TBF' (Time Between Failures), and 'Reliability'.

Output Layer: The output layer of the DNN has 3 neurons, each corresponding to one of the consequence attributes mentioned above.

Activation Functions: In this DNN, 'linear' activation is used in the hidden layer, which means the network performs a linear combination of the inputs. It's common to use 'linear' activation for regression tasks to predict continuous values.

Loss Function and Optimizer: The DNN is compiled using 'mean_squared_error' as the loss function, which is appropriate for regression tasks like predicting consequences. The 'adam' optimizer is used to minimize this loss.

This architecture is suitable for regression tasks where the goal is to predict numeric values (in this case, consequence attributes). The DNN takes the compressed representation of the input data, learns the underlying patterns, and outputs predictions for the three consequence attributes.

3.5 Proposed Baselines for Evaluation

In the comprehensive evaluation of software engineering methodologies, the inclusion of multiple baselines alongside the proposed methodology is crucial for providing a comparative analysis. Alternative baselines serve as benchmarks that help contextualize the performance and effectiveness of the proposed approach. By incorporating diverse baselines, including established methods and variations, the evaluation process gains depth and ensures a more robust understanding of the strengths and limitations of the proposed methodology. This comparative framework not only validates the proposed solution but also sheds light on its relative advantages and areas for improvement. Additionally, contrasting the proposed methodology with existing baselines allows for a more nuanced exploration of its uniqueness and applicability in different scenarios. Such a multi-baseline evaluation contributes to the broader field of software engineering research by offering insights into the versatility and generalizability of the proposed methodology, facilitating a more informed discussion on its potential impact and practical significance.

3.5.1 Mapping Requirements to two DNNs for predicting complexity and Consequence

Approach: In this approach, we could use the raw requirements data as inputs to separate DNNs for complexity and consequence prediction, effectively bypassing the need for feature engineering and complexity metrics as shown in Figure 13.

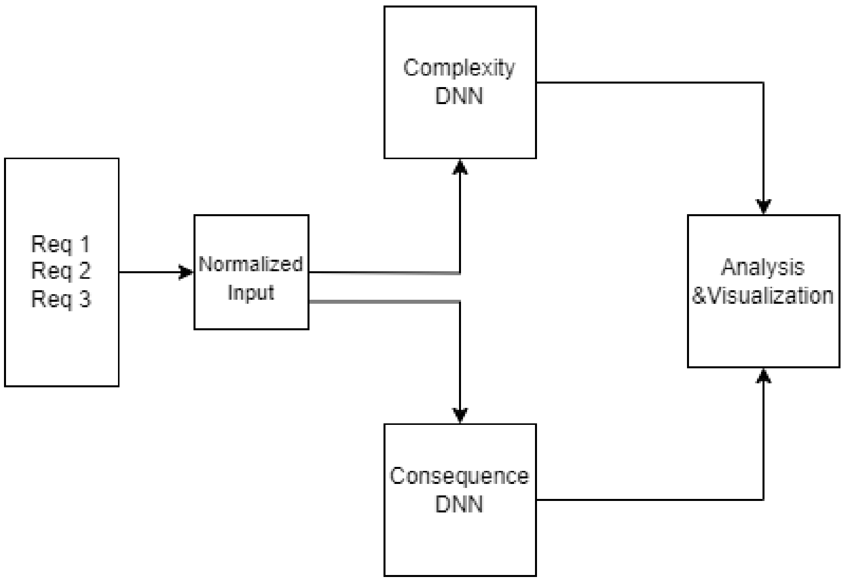


Figure 13:Overview of mapping requirements to DNNs

Reason for Not Using: This approach was not chosen likely due to the following reasons:

- Input Compatibility: The raw requirement data may not be directly compatible with traditional classification or regression models. DNNs are versatile in handling various data types, but preprocessing and feature engineering are often required to effectively use text or structured data-like requirements.

- Information Loss: By not using complexity metrics, you may lose valuable information that the metrics provide. Complexity metrics can capture specific characteristics of the requirements, which might be useful for quality attribute estimation. Removing them might result in the loss of relevant patterns.

3.5.2 Mapping Requirements to Consequence Directly with DNN:

Approach: This approach involves using a DNN to directly map requirements to consequence attributes without considering complexity metrics as shown in Figure 14.

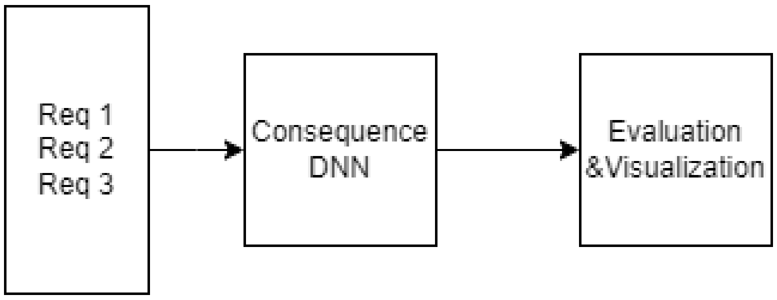


Figure 14: Mapping Requirements to Consequence

Reason for Not Using: It appears that this approach was not selected due to high prediction error. There could be several reasons for the high error:

- Complex Relationships: The relationships between requirements and consequences may be complex and nonlinear. Using complexity metrics as an intermediate step can help the model better capture these relationships.
- Missing Information: Complexity metrics provide additional information about the requirements that might be important for predicting consequences. Directly mapping requirements to consequences without this additional information could result in a loss of predictive power.

In summary, the chosen approach of using an autoencoder followed by separate DNNs for complexity and consequence prediction is likely the most effective way to handle high-dimensional, nonlinear data. It allows for the capture of complex patterns in the data and retains the valuable information provided by complexity metrics, ultimately resulting in more accurate predictions for software quality attributes.

4 Implementation

4.1 Use case: Robot Framework

Robot Framework is an open-source, keyword-driven test automation framework for acceptance testing and robotic process automation (RPA), acceptance testing and acceptance test-driven development (ATDD) [23]. It is designed to provide a simple, high-level syntax for writing test cases, making it an ideal choice for both testers and developers to create automated tests and automate repetitive tasks. The framework is versatile and has a wide range of use cases [23].

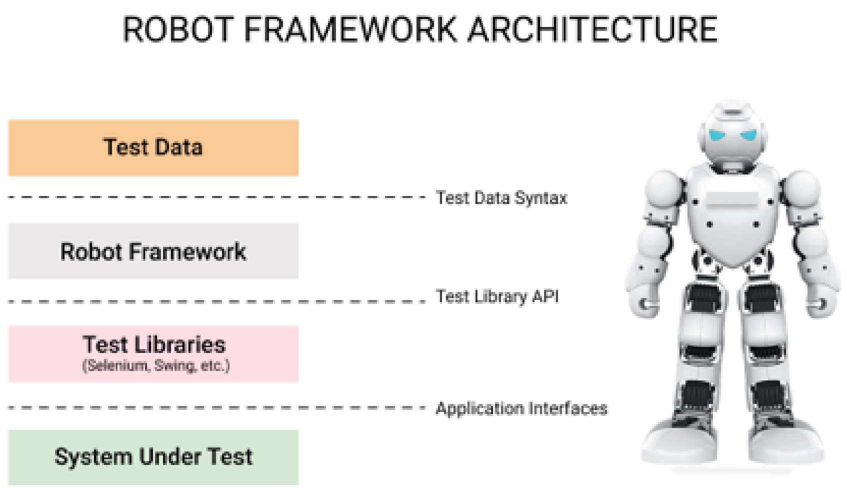


Figure 15: Robot Framework Architecture [24]

Robot Framework is open and extensible. The architecture of Robot Framework is shown in Figure 15. Robot Framework can be integrated with virtually any other tool to create powerful and flexible automation solutions. Robot Framework is free to use without licensing costs.

Robot Framework has an easy syntax, utilizing human-readable keywords. Its capabilities can be extended by libraries implemented with Python, Java or many other programming languages. Robot Framework has a rich ecosystem around it, consisting of libraries and tools that are developed as separate projects [23].

4.2 Requirement

In software development, a commit message is a description of the changes made to the source code in a single commit. These messages are used to document the purpose, scope, and details of each code change. The commit messages are typically created by developers when they commit their code changes to a version control system like Git. This thesis involves extracting or deriving software requirements from these commit messages. This is a valuable and innovative way to bridge the gap between code changes and the underlying project requirements. It can improve transparency, traceability, and collaboration in software development projects while presenting some challenges related to the quality and structure of commit messages.

4.3 Dataset

The dataset is categorized into three primary sections: Requirements, Complexity, and Consequences as shown in Table 2. The Requirements section, referred to as Title Vector, encompasses tokenized data processed as vectors derived from strings. Within the Complexity section, two subsections exist, comprising lines of code and cyclomatic complexity. The Consequences section further divides into three subsections, incorporating the number of calls, TBF (time between failures), and reliability. The dataset comprises a total of 3041 rows of data, serving as input for the autoencoder.

Table 2: Dataset sample

Serial Number	Requirements		Complexity		Consequence		
	Original Title	Title_Vector	Lines of Code	Cyclomatic Complexity	Number of Calls	TBF	Reliability
1	Remove deprecated attributes in result model	8.45413208e-02 2.31666565e-02 4.81465645e-02 8.02612305e-02	127	11	3	0	1
2	Support type aliases in formats ``’list[int]’`` and ``’int | float’`` in argument conversion	5.8851249e-02 - 4.94995117e-02 7.41170272e-02 2.05240890e-01	127	11	2	0	1
3	User keyword teardown missing from running model JSON schema	5.80851249e-02 - 4.94995117e-02 7.41170272e-02 2.05240890e-01	127	11	5	0	1
4	Remote: Enhance `datetime`, `date` and `timedelta` conversion	0.10810547 0.02653809 - 0.06296997 0.01600342 - 0.04169922 0.01412964	127	11	2	2	0.972321
…	…	….	..	..	..	..	..
							TOTAL = 3041

4.4 Libraries used for implementation

1) **Pandas:**

Pandas is a data manipulation and analysis library for Python. It provides data structures and functions to work with structured data, making it easier to handle and analyze data [25].

Key Features:

- DataFrames: Pandas introduces the DataFrame, which is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labelled axes.
- Data Cleaning: Pandas offers tools for data cleaning, transformation, and handling of missing data.
- Data Import/Export: It supports reading data from various file formats (CSV, Excel, SQL databases) and exporting data as well.

Use Case: Pandas is commonly used in data analysis, data preprocessing for machine learning, and data wrangling tasks.

2) **NumPy:**

NumPy, short for Numerical Python, is a fundamental package for scientific computing with Python. It provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays [26].

Key Features:

- Multi-Dimensional Arrays: NumPy arrays are efficient for handling large datasets and performing array operations.
- Mathematical Functions: NumPy includes a wide range of mathematical functions for operations like linear algebra, Fourier analysis, and random number generation.

Use Case: NumPy is a foundational library for numerical and scientific computing, commonly used in data analysis, machine learning, and scientific research.

3) **scikit-learn (sklearn):**

Scikit-learn is a machine-learning library for Python. It provides simple and efficient tools for data mining and data analysis, including various classification, regression, clustering, and dimensionality reduction algorithms [26].

Key Features:

- Simple and Consistent API: Scikit-learn offers a consistent interface for working with various machine learning models and tasks.
- Wide Range of Algorithms: It includes a broad set of algorithms for supervised and unsupervised learning.

- Model Evaluation: Scikit-learn provides tools for model evaluation, model selection, and hyperparameter tuning.

Use Case: Scikit-learn is widely used in machine learning, data analysis, and building predictive models.

4) **TensorFlow:**
TensorFlow is an open-source machine learning framework developed by Google. It provides a comprehensive ecosystem of tools, libraries, and community resources for developing machine learning and deep learning models [27].

Key Features:
- Computational Graphs: TensorFlow models are defined using computational graphs, making it highly suitable for deep learning.
- Flexibility: It supports various levels of abstraction, from low-level mathematical operations to high-level APIs for model building.
- Deployment: TensorFlow enables easy deployment of machine learning models to various platforms.

Use Case: TensorFlow is widely used in deep learning, neural networks, and building complex machine learning models.

5) **Matplotlib:**
Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It provides tools for creating a wide range of plots, graphs, and charts [26].

Key Features:
- Customizable Plots: Matplotlib allows you to customize every aspect of your visualizations.
- Publication-Quality Output: It is suitable for creating high-quality figures for publications and presentations.

Use Case: Matplotlib is commonly used for data visualization, scientific plotting, and creating informative graphics.

4.5 Process Pipeline

The process pipeline for the estimation and evaluation of complexity and consequence metrics, utilizing an innovative approach is centered around the autoencoder as shown in Figure 16. In the initial phase, the autoencoder's reconstruction performance is assessed through the calculation of Mean Squared Error (MSE) for consequence reconstruction, presenting the result as a percentage. Additionally, a new model is crafted to extract the compressed representation from the trained autoencoder, catering to both training and testing datasets.

Moving forward, a pivotal component of the pipeline involves the creation of a bijective mapping model. This model takes the compressed data as input and predicts complexity metrics, specifically Lines of Code and Cyclomatic Complexity. The bijective model is meticulously compiled, employing the Adam optimizer and utilizing Mean Squared Error (MSE) as the loss function. Subsequently, the model undergoes training on the compressed training data and validation on the compressed test data.

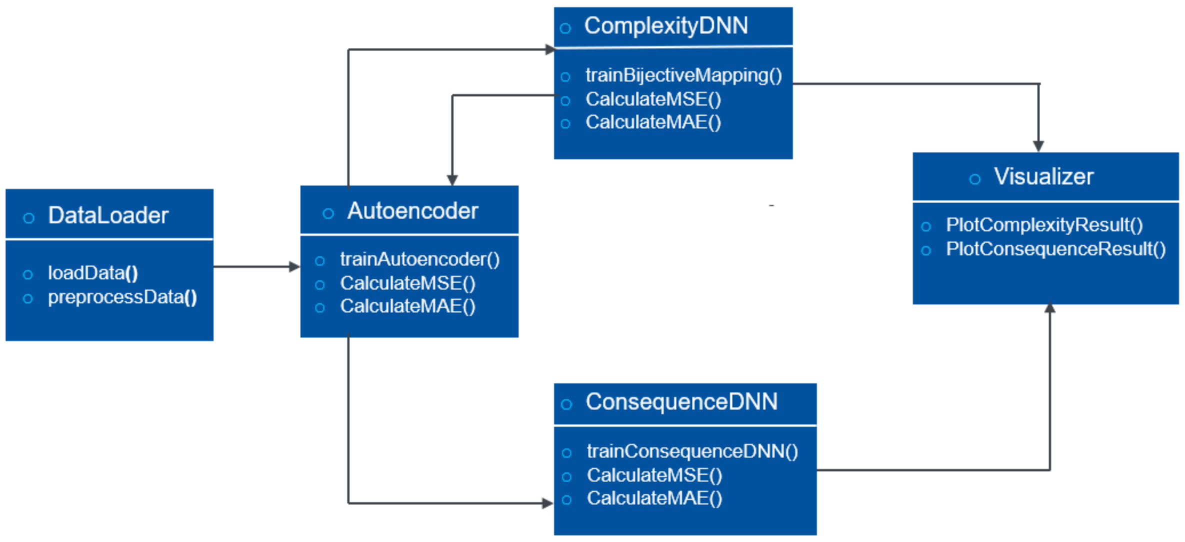


Figure 16: Process pipeline of proposed architecture

The subsequent phase delves into the evaluation of complexity predictions. The bijective mapping model is employed to predict complexity values on the validation set. The assessment includes the calculation of both Mean Squared Error (MSE) and Mean Absolute Error (MAE) for complexity prediction. These metrics are presented in both absolute values and percentages, providing a comprehensive understanding of the model's accuracy relative to the range of complexity values.

Transitioning to the consequence prediction aspect, the pipeline introduces a Deep Neural Network (DNN) designed for predicting consequences. The DNN is configured with the Adam optimizer and employs Mean Squared Error (MSE) as the chosen loss function. The model undergoes training on the compressed training data and subsequent validation on the compressed test data.

The final phase of the pipeline involves the evaluation of consequence predictions. The trained DNN is employed to predict consequences on the validation set. Evaluation metrics, including Mean Squared Error (MSE) and Mean Absolute Error (MAE), are calculated and expressed as both absolute values and percentages. These metrics provide valuable insights into the model's performance and its ability to generalize effectively to previously unseen data. Altogether, this integrated pipeline showcases the efficacy of autoencoders, bijective mapping, and DNNs in accurately estimating and predicting complexity and consequence metrics in the realm of software requirements.

5 Results and Evaluation

In this section, we conduct an in-depth analysis and present the outcomes of our proposed architectural framework. Here, we juxtapose the results achieved through our innovative methodology with those obtained from two alternative baseline approaches: 1) Utilizing DNNs for both complexity and consequences independently, without the incorporation of autoencoders. 2) Directly mapping requirements to consequences without the intermediary step of dimensionality reduction and feature learning through autoencoders.

5.1 Proposed Methodology: Autoencoder with DNN

This methodology encompasses outcomes from three key components of the architecture. The initial segment entails the assessment of Autoencoder's performance in terms of original data versus reconstruction. The second segment delves into the results obtained from the bijective model, responsible for predicting complexity. The final segment focuses on the consequence prediction results. For each of these components, both Mean Squared Error (MSE) and Mean Absolute Error (MAE) metrics are computed and reported.

5.1.1 Autoencoder actual data vs Reconstruction graph

The initial segment of the methodology involves assessing the performance of an autoencoder in terms of compressing and reconstructing the original data. An autoencoder is a type of neural network designed to encode data into a lower-dimensional representation and then reconstruct it back to the original format. This process helps in capturing essential features of the data and reducing its dimensionality. To visually assess the performance, a graph is shown in figure 17 where the x-axis represents complexity and consequence, and the y-axis represents the mean value. Two sets of data are plotted on the graph, original data and reconstructed data. If the reconstructed points closely match the original points, it suggests that the Autoencoder effectively captured the important features during compression and accurately reconstructed the data. In summary, the initial segment involves assessing the Autoencoder's ability to compress and reconstruct the original data, and graphical representation with MSE and MAE metrics provides a comprehensive evaluation of the reconstruction performance.

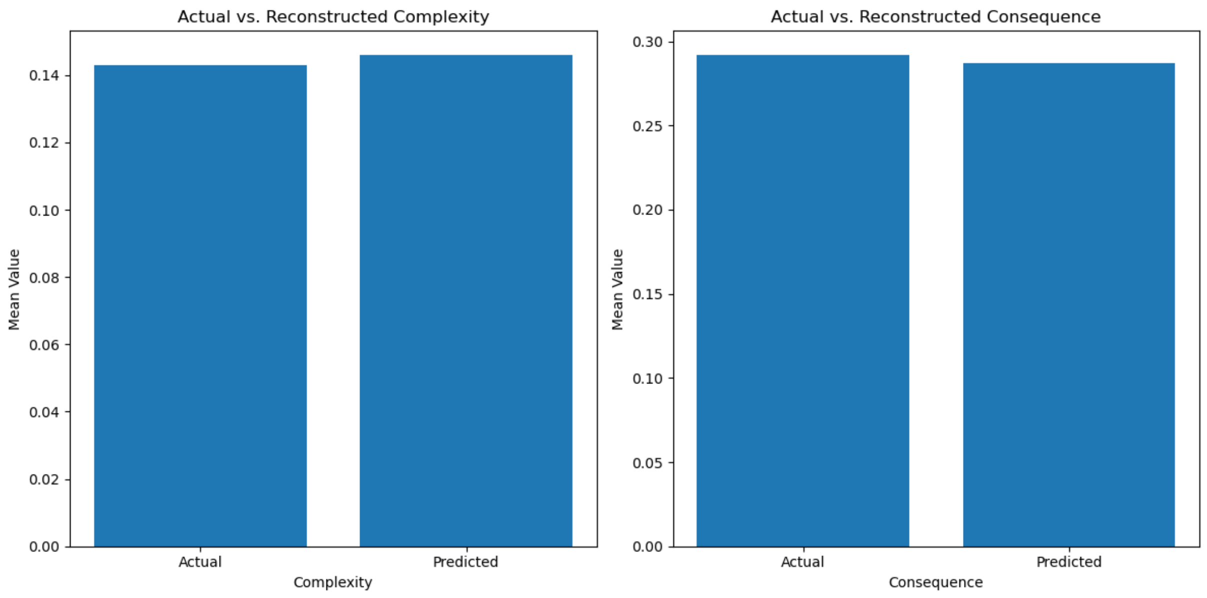


Figure 17: Autoencoder actual vs reconstruction graph

Table 3: Autoencoder reconstruction result

Evaluation Metrics	MSE	MAE
Complexity	4.93%	14.3%
Consequence	4.15%	12.69%
Total	0.12%	3.53%

In the graphical representation of our Autoencoder's performance, an impressive level of accuracy was observed. The comparison between the actual data and the reconstructed data showcases the precision of the model as shown in Table 3. The total Mean Squared Error, calculated as a percentage, stands at a remarkably low 0.12%, indicating an exceptionally close fit. Additionally, the Mean Absolute Error expressed as a percentage, demonstrates an equally impressive accuracy, with a value of only 3.53%. Further scrutinizing our results, we find that the Reconstruction Mean Squared Error for Complexity amounts to 4.93%, while for Consequences, it stands at 4.15%. These findings emphasize the robustness of our Autoencoder, underscoring its capacity to faithfully reproduce the input data while maintaining an exceptionally high level of fidelity.

5.1.2 Bijective DNN Model for Predicting Complexity

The second segment focuses on the outcomes derived from a bijective model, a customized Deep Neural Network (DNN) architecture responsible for predicting complexity. In this context, the bijective mapping model takes the compressed data generated by the compression layer as its input and predicts the corresponding complexity values, including Lines of Code and Cyclomatic Complexity, as its output. The uniqueness of this mapping lies in its bijective nature, allowing it to seamlessly navigate between the compressed data and complexity values in both directions. To visually assess the performance of the bijective model, a graph is created where the x-axis features a bar chart representing Lines of Code and Cyclomatic Complexity. In Figure 18, the y-axis represents the mean value. Two sets of bars are plotted on the graph one for the original data and one for the reconstructed data. If the bars for the reconstructed data closely align with the bars for the original data, it indicates that the bijective model effectively predicts complexity values based on the compressed data.

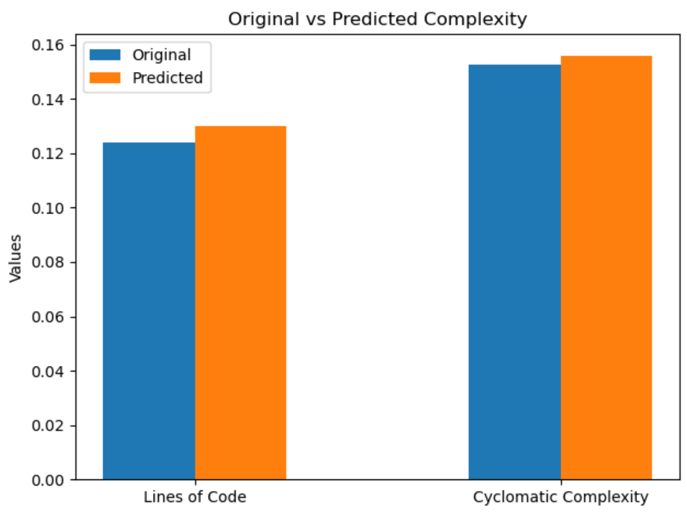


Figure 18: Complexity original vs Predicted graph

Table 4: Complexity Evaluation for Proposed Methodology

Evaluation Metrics	MSE	MAE
Lines Of Code	5.01%	14.41%
Cyclomatic Complexity	4.86%	14.19%
Total	4.93%	14.3%

In the evaluation of our Bijective DNN model's performance concerning the prediction of complexity attributes, we assess its accuracy through key metrics as shown in Table 4. The Total Mean Squared Error expressed as a percentage, is calculated at 4.93%. This value provides insight into the overall precision of our model, demonstrating a remarkable alignment between the original and predicted complexity attributes. Similarly, the Total Mean Absolute Error, at a low 14.30%, underscores the model's ability to maintain accuracy in its predictions. Furthermore, when we examine individual complexity attributes, the Mean Squared Error for Lines of Code is determined to be 5.01%, while the Mean Absolute Error for this attribute is measured at 14.41%. Likewise, for Cyclomatic Complexity, we find a Mean Squared Error of 4.86% and a Mean Absolute Error of 14.19%. These results highlight the robustness of our Bijective DNN model, indicating its proficiency in accurately predicting complexity attributes and maintaining a high level of fidelity.

5.1.3 DNN Model for Predicting Consequences

The concluding segment centers on the outcomes of consequence prediction, utilizing input from the compression layer of the Autoencoder. The neural network architecture employed in this scenario is a feedforward neural network, characterized by a single hidden layer. The primary focus of this network is to predict consequences, and the prediction involves key parameters such as the number of calls, Time Between Failures (TBF), and Reliability. Similar to the previous segments, the evaluation involves a graphical representation comparing original data with reconstructed data. This graph in Figure 19, features a bar chart on the x-axis representing the number of calls, TBF, and Reliability, while the y-axis indicates the mean value. This comparison provides insights into the accuracy of consequence predictions made by the neural network, giving a visual representation of how well the model performs in predicting key parameters based on the compressed data from the Autoencoder's compression layer.

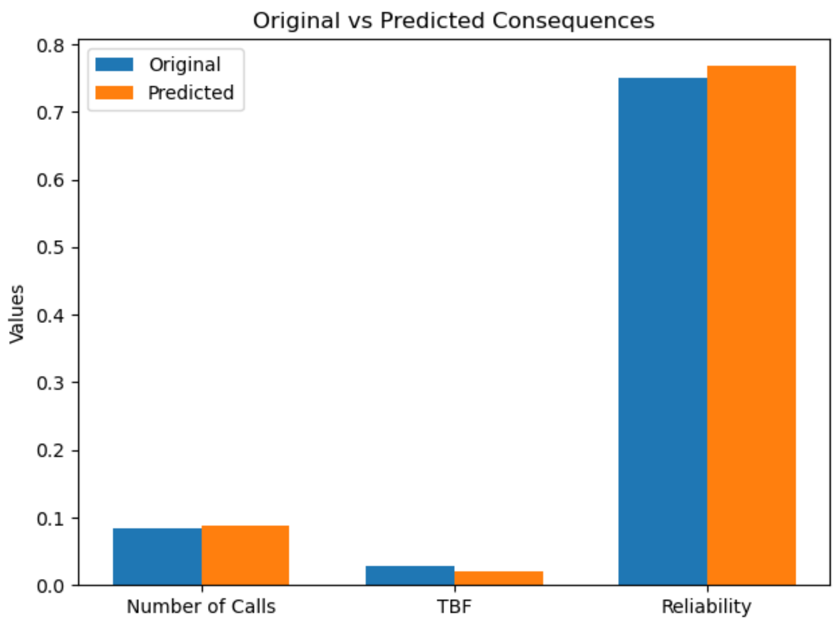


Figure 19: Consequence original vs Predicted graph

Table 5: Consequence Evaluation for Proposed Methodology

Evaluation Metrics	MSE	MAE
Number of calls	1.37%	10.58%
TBF	0.71%	4.62%
Reliability	10.95%	27.26%
Total	4.15%	12.69%

In the assessment of our DNN Model's performance in predicting consequences, we focus on key metrics that shed light on its accuracy and reliability as shown in Table 5. The Total Consequence Prediction Mean Squared Error (MSE), expressed as a percentage, is measured at a remarkable 4.15%. This percentage underscores the high level of precision and consistency in the predictions made by our model concerning a range of consequence attributes. Similarly, the Total Consequence Prediction Mean Absolute Error (MAE) is impressively low, at 12.69%, reinforcing the model's ability to maintain a high degree of fidelity in its predictions.

Further granularity is provided by evaluating the model's performance on individual consequence attributes. For 'Number of Calls,' the Mean Squared Error is notably low at 1.37%, and the Mean Absolute Error is measured at 10.58%. This indicates the model's capability to make accurate predictions while maintaining a low margin of error. Additionally, for 'TBF' (Time Between Failures), our model achieves an even lower Mean Squared Error of 0.71% and an impressively low Mean Absolute Error of 4.62%, illustrating its proficiency in capturing the nuanced patterns and trends within the data.

Lastly, for 'Reliability,' we note a Mean Squared Error of 10.95% and a Mean Absolute Error of 27.26%. While the MSE for this attribute is higher in comparison to the others, the model's capacity to predict 'Reliability' remains robust, making it a valuable tool for decision-making in complex systems. These results collectively highlight the strength and accuracy of our DNN Model in predicting a spectrum of consequence attributes with precision and efficiency.

5.2 Baseline 1: DNN for complexity and consequences

This baseline for Complexity and Consequence Prediction employs a Deep Neural Network (DNN) without the use of an Autoencoder. This foundational approach comprises two primary sections. The first section centers on predicting complexity, while the second section is dedicated to forecasting consequences. In each of these segments, the evaluation involves the computation and presentation of Mean Squared Error (MSE) and Mean Absolute Error (MAE) metrics. This baseline serves as a fundamental framework for assessing the accuracy of predictions regarding complexity and consequences, providing insights into the performance of the DNN model in the absence of an Autoencoder.

5.2.1 Complexity prediction

The initial section of the study focuses on the task of predicting complexity. A key aspect of this analysis involves comparing the original data with the reconstructed data through a graphical representation. In graph shown in Figure 20, the y-axis signifies the mean value, while the x-axis features a bar chart detailing the values of lines of code and cyclomatic complexity. The comparison includes both the original and the reconstructed data for each of these complexity metrics. This graphical illustration serves as a comprehensive visual tool to assess the accuracy of the complexity predictions, shedding light on how well the model captures and reproduces the original complexity features.

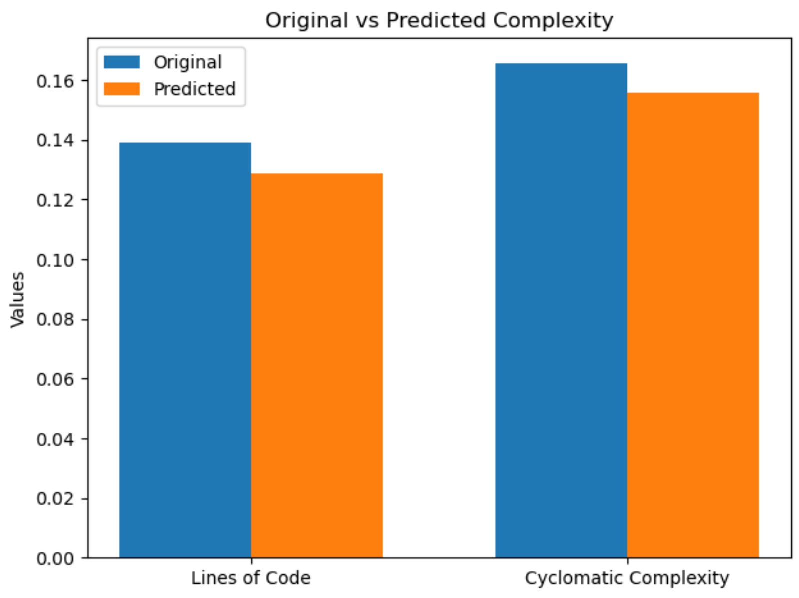


Figure 20: Complexity original vs Predicted graph

When evaluating the performance of our Deep Neural Network (DNN) for complexity prediction without the integration of the autoencoder, we observed notable results as shown in Table 6. The Total Complexity Prediction Mean Squared Error (MSE), expressed as a percentage, stands at 6.32%. Additionally, the Total Complexity Prediction Mean Absolute Error (MAE) is registered at 15.68%. These figures offer insights into the DNN's ability to predict complexity attributes, where precision and accuracy remain important.

Table 6: Complexity Evaluation for Baseline1

Evaluation Metrics	MSE	MAE
Lines Of Code	6.46%	15.86%
Cyclomatic Complexity	6.18%	15.51%
Total	6.32%	15.68%

Delving deeper into the specifics, the Cyclomatic Complexity Prediction results are distinguished by a Mean Squared Error of 6.18% and a Mean Absolute Error of 15.51%. Simultaneously, the Lines of Code Prediction results feature a Mean Squared Error of 6.46% and a Mean Absolute Error of 15.86%. These findings underscore the capabilities of the DNN model in predicting complexity metrics. While these results provide a solid foundation for complexity prediction, our integration of the autoencoder, as previously mentioned, has shown the potential to further enhance prediction accuracy by reducing dimensionality and eliminating noise.

5.2.2 Consequence Prediction

The subsequent segment is specifically focused on predicting consequences. Within this analysis, a graphical representation is employed to compare the original data with the reconstructed data. In this graph, the y-axis represents the mean value, while the x-axis incorporates a bar chart displaying the values of the number of calls, Time Between Failures (TBF), and Reliability. This comparison encompasses both the original and the reconstructed data for each of these consequence metrics. This graphical visualization in Figure 21 serves as a comprehensive tool to evaluate the precision of consequence predictions, providing insights into how effectively the model anticipates and replicates the original characteristics related to consequences.

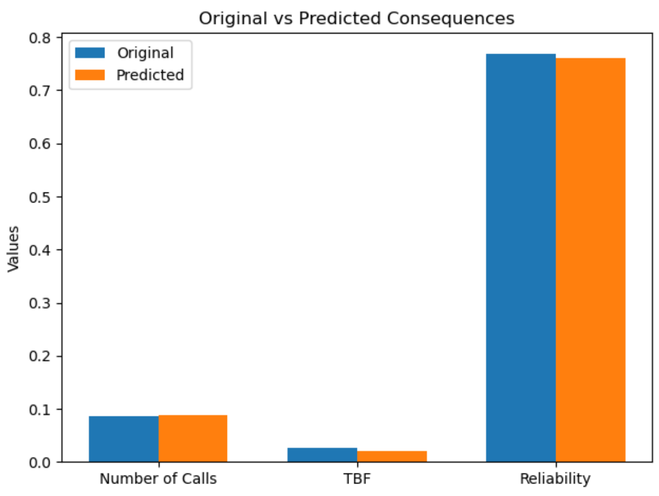


Figure 21: Consequence original vs Predicted graph for Baseline 1

The evaluation of our Deep Neural Network (DNN) for consequence prediction, conducted independently of the autoencoder, has revealed important insights into the model's performance as shown in Table 7. The Total Consequence Prediction Mean Squared Error (MSE), reported as a percentage, stands at 4.30%. Furthermore, the Mean Absolute Error (MAE) for the total consequence prediction is noted at 12.77%. These statistics offer valuable indications of the DNN's proficiency in predicting consequence attributes, which is essential for various applications.

Digging into the specifics of consequence prediction, the Number of Calls demonstrates a prediction MSE of 11.00% and an MAE of 27.13%. In the case of the Time Between Failures (TBF), the MSE is remarkably lower at 1.14%, and the MAE is 7.66%. Moreover, for Reliability prediction, the MSE is a mere 0.88%, while the MAE is a notable 4.17%. These findings highlight the DNN's efficacy in forecasting various consequence attributes.

The evaluation of our Deep Neural Network (DNN) for consequence prediction, conducted independently of the autoencoder, has revealed important insights into the model's performance. The Total Consequence Prediction Mean Squared Error (MSE), reported as a percentage, stands at 4.30%. Furthermore, the Mean Absolute Error (MAE) for the total consequence prediction is noted at 12.77%. These statistics offer valuable indications of the DNN's proficiency in predicting consequence attributes, which is essential for various applications.

Digging into the specifics of consequence prediction, the Number of Calls demonstrates a prediction MSE of 11.00% and an MAE of 27.13%. In the case of the Time Between Failures (TBF), the MSE is remarkably lower at 1.14%, and the MAE is 7.66%. Moreover, for Reliability prediction, the MSE is a mere 0.88%, while the MAE is a notable 4.17%. These findings highlight the DNN's efficacy in forecasting various consequence attributes.

Table 7: Consequence Evaluation for Baseline1

Evaluation Metrics	MSE	MAE
Number of calls	11.00%	27.13%
TBF	1.14%	7.66%
Reliability	0.88%	4.17%
Total	4.3%	12.77%

While these results are promising, it is important to remember that our integration of the autoencoder provides an avenue for further enhancement in prediction accuracy, particularly in scenarios where data volume is substantial and contains noise and irrelevant features. The autoencoder's ability to reduce dimensionality and capture essential patterns in the data has shown the potential to refine the precision and robustness of our consequence predictions.

5.3 Baseline 2: Mapping Requirements to Consequences

The foundational approach in this context employs a Deep Neural Network (DNN) to directly connect requirements to consequence attributes, bypassing the consideration of complexity metrics and the use of an autoencoder. The evaluation of this baseline methodology involves a graphical representation comparing the original data to the reconstructed data. In Figure 22, the y-axis signifies the mean value, while the x-axis presents a bar chart illustrating the values of the number of calls, Time Between Failures (TBF), and Reliability. This comparison encompasses both the original and the reconstructed data for each of these consequence attributes. This graphical representation serves as a comprehensive tool for assessing the accuracy of the DNN in directly mapping requirements to consequence attributes, offering insights into how well the model captures and reproduces the original characteristics related to consequences without relying on complexity metrics or an autoencoder.

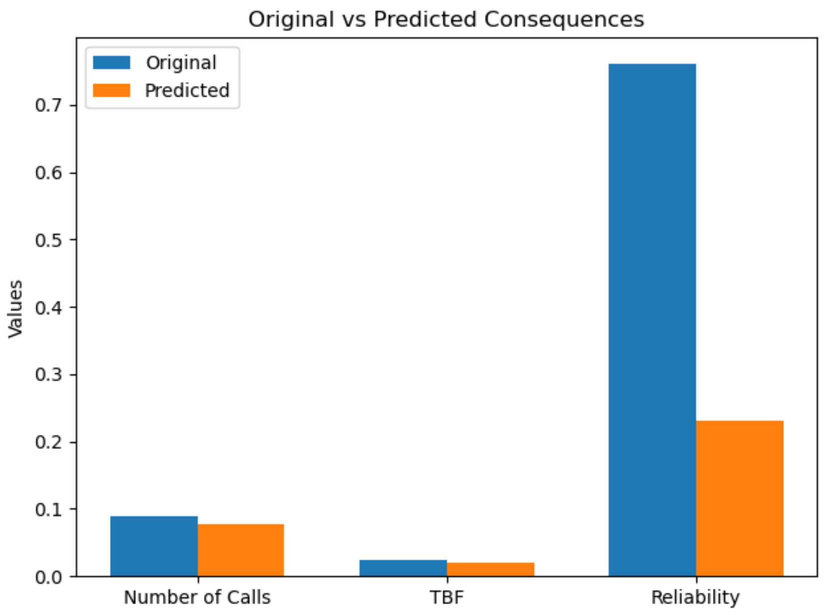


Figure 22: Consequence Original vs Predicted graph for Baseline 2

The evaluation of our Deep Neural Network (DNN) for consequence prediction, conducted independently of the autoencoder, has shed light on its performance characteristics. In this scenario, the Total Consequence Prediction Mean Squared Error (MSE), expressed as a percentage, is measured at 13.78% as shown in Table 8. Similarly, the Mean Absolute Error (MAE) for total consequence prediction is registered at 23.09%. These metrics provide essential insights into the accuracy and robustness of our DNN model in predicting various consequence attributes.

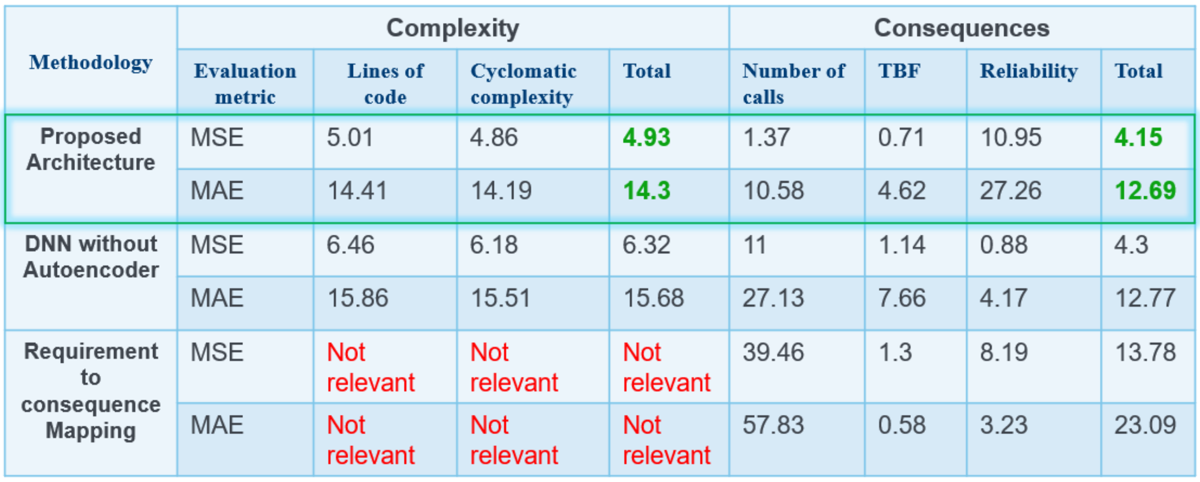
When we delve into the specifics of consequence prediction, the results unveil distinct trends. For Number of Calls prediction, the MSE is relatively high at 39.46%, and the MAE reaches 57.83%. However, the Time Between Failures (TBF) prediction yields a much lower MSE at 1.30%, coupled with an MAE of 8.19%. Remarkably, for Reliability prediction, the MSE is a mere 0.58%, and the MAE is a notably low 3.23%. These observations underscore the versatility and capabilities of our DNN in forecasting different consequence attributes with varying degrees of precision.

Table 8: Consequence Evaluation for Baseline 2

Evaluation Metrics	MSE	MAE
Number of calls	39.46%	23.09%
TBF	1.3%	8.19%
Reliability	0.58%	3.23%
Total	13.78%	23.09%

It is essential to consider these outcomes in the broader context of our research, as they demonstrate the significance of the autoencoder's integration. By incorporating the autoencoder to preprocess and distil the input data, we aim to improve the accuracy and robustness of consequence predictions, particularly when faced with large datasets containing noise and irrelevant features.

5.4 Comparison of Results



Methodology	Complexity				Consequences			
	Evaluation metric	Lines of code	Cyclomatic complexity	Total	Number of calls	TBF	Reliability	Total
Proposed Architecture	MSE	5.01	4.86	**4.93**	1.37	0.71	10.95	**4.15**
	MAE	14.41	14.19	**14.3**	10.58	4.62	27.26	**12.69**
DNN without Autoencoder	MSE	6.46	6.18	6.32	11	1.14	0.88	4.3
	MAE	15.86	15.51	15.68	27.13	7.66	4.17	12.77
Requirement to consequence Mapping	MSE	Not relevant	Not relevant	Not relevant	39.46	1.3	8.19	13.78
	MAE	Not relevant	Not relevant	Not relevant	57.83	0.58	3.23	23.09

Figure 23: Comparison of Results

The proposed methodology, which integrates the use of both an autoencoder and a DNN, outperforms the other two approaches: DNN for complexity and consequences without an autoencoder, and Mapping Requirements to Consequences as shown in Figure 23. The superior performance of the proposed methodology can be attributed to its ability to leverage the feature learning and dimensionality reduction capabilities of the autoencoder, providing a more effective representation of the input data. This enriched representation contributes to enhanced accuracy in both complexity and consequence predictions when compared to the alternative methods. The combination of the autoencoder's preprocessing power and the DNN's predictive capabilities proves to be a more robust and accurate approach for estimating software quality attributes.

6 Conclusion

This research project explored the application of autoencoders and deep neural networks (DNNs) to estimate software quality attributes, specifically focusing on complexity metrics and their implications on software reliability and maintainability. It yielded several important findings.

Having an autoencoder to extract meaningful feature representations from high-dimensional software requirements data provided an effective means of reducing dimensionality and capturing essential information. Also adding separate DNN models for predicting software complexity and its consequences added a lot of benefits. The multi-step approach of using an autoencoder for feature extraction followed by DNNs for prediction demonstrated promising results, with competitive predictive accuracy. Comparing this method with alternative approaches, such as directly using DNNs and mapping requirements to consequences without using complexity metrics revealed its superiority in terms of prediction accuracy.

The work highlights the significance of leveraging complexity metrics to enhance the prediction of software quality attributes. The use of autoencoders for feature extraction contributes to improved model performance and reliability estimation.

7 Future Scope

This research has demonstrated that the multi-step approach of utilizing auto-encoders for feature extraction, followed by separate DNNs for complexity and consequence prediction, shows great promise. However, this work underscores the significance of incorporating complexity metrics for enhanced software quality attribute prediction, particularly concerning reliability and maintainability.

In the realm of future work, exploring the integration of diverse data sources, experimenting with advanced neural network architectures, and developing hybrid models that merge autoencoders with other unsupervised learning techniques would be the field of curiosity [28]. Real-time predictions, benchmarking, and industry applications are also avenues worth investigating [28]. Furthermore, the development of explanatory models could contribute to the transparency and comprehensibility of software quality predictions. These proposed directions offer a roadmap for researchers and practitioners aiming to advance the field of software quality attribute estimation.

8 Bibliography

[1] A. Baghdasaryan, "Medium," 2018. [Online]. Available: https://medium.com/fintegro-company-inc/why-is-software-quality-assurance-important-bd5f0f2bcd9.

[2] S. V. P. W. G. D.I. De Silva, "Software Complexity Metrics for Emerging Technologies: Challenges and Opportunities," pp. 1-5, 2023.

[3] C. K. Pasindu Madhuwantha, "The Effectiveness of Software Complexity Metrics in Predicting Software Performance Bottlenecks," pp. 1-6, 2023.

[4] K. M. Kamaljit Kaur, "Static and Dynamic Complexity Analysis of Software Metrics," vol. 3, pp. 1-3, 2009.

[5] A. authors, "DEEP LEARNING-BASED SOURCE CODE COMPLEXITY," 2023.

[6] geeksforgeeks, 2023. [Online]. Available: https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/.

[7] N. P. T. S. Sergiy Prykhodko, "A Statistical Evaluation of The Depth of Inheritance Tree Metric," pp. 1-14, 2021.

[8] N. K. a. A. S. Mrinal Kanti Debbarma, "Static and Dynamic Software Metrics Complexity Analysis," pp. 1-7, 2012.

[9] K. Alberts, "Comparison of different types of auto-encoders for data," pp. 1-9, 2021.

[10] T.-Y. H. P.-C. C. a. J.-H. W. Chin-Wei Tien, "Using Autoencoders for Anomaly Detection and Transfer," pp. 1-14, 2021.

[11] A. Prakash, "Open Genus," 2016. [Online]. Available: https://iq.opengenus.org/types-of-autoencoder/.

[12] J. A. B. J. A.-M. Maria Teresa Garcia-Ordas, "Determining the severity of Parkinson's disease," pp. 1-16, 2021.

[13] I. k. a. A. Mishra, "Automated Classification Method for Early Diagnosis of Alopecia Using Machine Learning," pp. 1-7, 2018.

[14] S. E. Elçi, "Medium," 2020. [Online]. Available: https://medium.com/@serkanemreelci/denosing-images-with-autoencoders-1b2833441338.

[15] A. Ponomareva, "Medium," 2021. [Online]. Available: https://medium.com/nico-lab/disentangled-variational-autoencoders-for-brain-ct-slices-723a21e49813.

[16] L. I. P. P. Pintelas E, "A Convolutional Autoencoder Topology," 2021.

[17] A. Fadaeinejad, "Anomaly Detection in Images using Deep Encoder-Decoder Models," pp. 1-7, 2020.

[18] P. J. F. G. T. H. Michael Greenacre, "Principal Component Analysis," pp. 1-25, 2022.

[19] Baeldung, August 2023. [Online]. Available: https://www.baeldung.com/cs/normalizing-inputs-artificial-neural-network.

[20] J. Jordan, March 2018. [Online]. Available: https://www.jeremyjordan.me/autoencoders/.

[21] S. N. J. V. H. K. Kaliprasad C S, "A Review on Neural Networks and its Applications," vol. 40, no. 2, pp. 1-13, 2023.

[22] J. Dancker, "An introduction to RNN, LSTM, and GRU and their implementation," December 2022. [Online]. Available: https://towardsdatascience.com/a-brief-introduction-to-recurrent-neural-networks-638f64a61ff4.

[23] Robot Framework, February 2017. [Online]. Available: https://robotframework.org/.

[24] Payoda Technology Inc, "Medium," July 2021. [Online]. Available: https://payodatechnologyinc.medium.com/introduction-to-functional-web-automation-with-robot-framework-127197af299d.

[25] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics," pp. 1-10, 2011.

[26] L. G. S. S. Lavanya Addepalli, "Assessing the Performance of Python Data Visualization Libraries: A Review," pp. 1-13, 2023.

[27] Google, "first-steps-with-tensorflow," November 2015. [Online]. Available: https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit.

[28] J. L. Y. J. H. H. Zhenyu Yuan, "Hybrid-DNNs: Hybrid Deep Neural Networks for Mixed Inputs," pp. 1-14, 2005.

[29] R. L. S. JOSEPH K. KEARNEY, "SOFTWARE COMPLEXITY MEASUREMENT," vol. 29, pp. 1-7, 1986.

Declaration of Compliance

I hereby declare to have written this work independently and to have respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as images, drawings, text passages) are used in this work, I declare that these materials are referenced accordingly (e.g. quote, source) and, whenever necessary, consent from the author to use such materials in my work has been obtained.

Signature: [illegible] Stuttgart, on the 27.11.2023