

Comparison of Deep Neural Networks and Regression Models for Software Effort Estimation

Bc. Ján Michalov

Master's thesis
2025



Tomas Bata University in Zlín
Faculty of Applied Informatics

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Ján Michalov
Osobní číslo:	A23408
Studijní program:	N0613A140022 Informační technologie
Specializace:	Kybernetická bezpečnost
Forma studia:	Kombinovaná
Téma práce:	Srovnání hlubokých neuronových sítí a regresních modelů pro odhadování softwarového úsilí
Téma práce anglicky:	Comparison of Deep Neural Networks and Regression Models for Software Effort Estimation

Zásady pro vypracování

- Provedte literární rešerši predikčních metod pro odhad vývojového úsilí.
- Definujte metriky pro hodnocení přesnosti modelů a popište testovací data.
- Navrhněte experiment s využitím modelů hlubokého učení a regresí.
- Porovnejte testované modely a diskutujte jejich výkon.
- Navrhněte praktická doporučení a směry pro budoucí výzkum v oblasti predikce softwarového úsilí.

Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

1. JAMES, Gareth; WITTEN, Daniela; HASTIE, Trevor a TIBSHIRANI, Robert. *An introduction to statistical learning: with applications in R*. Second edition. *Springer texts in statistics*. New York: Springer, [2021]. ISBN 978-1-0716-1417-4
2. HASTIE, Trevor; TIBSHIRANI, Robert a FRIEDMAN, J. H. *The elements of statistical learning: data mining, inference, and prediction*. 2nd ed. *Springer series in statistics*. New York: Springer, c2009. ISBN 978-0-387-84857-0
3. KELLEHER, John D.; NAMEE, Brian Mac a D'ARCY, Aoife. *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. Cambridge: The MIT Press, [2015]. ISBN 978-0-262-02944-5
4. GOODFELLOW, Ian; BENGIO, Yoshua a COURVILLE, Aaron. *Deep learning. Adaptive computation and machine learning series*. Cambridge, Massachusetts: The MIT Press, [2016]. ISBN 978-0-262-03561-3
5. MURPHY, Kevin P. *Machine learning: a probabilistic perspective. Adaptive computation and machine learning*. Cambridge: MIT Press, [2012]. ISBN 978-0-262-01802-9
6. MCCONNELL, Steve. *Software estimation: demystifying the black art*. Redmond: Microsoft Press, c2006. ISBN 0-7356-0535-1
7. BARRYBOEHM, Barry a CHRIS, Abts. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000. ISBN 978-0-137025763

Vedoucí diplomové práce: **doc. Ing. Radek Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce zkoumá efektivitu hlubokých neuronových sítí ve srovnání s tradičními regresními modely pro odhad softwarového úsilí. Po úvodním přehledu prediktivních technik a metrik hodnocení byla navržena strukturovaná experimentální studie porovnávající čtyři regresní modely — lineární regresi, polynomiální regresi, Lasso a Ridge — s dopřednou (feedforward) neuronovou sítí.

Regresní modely prošly laděním pomocí grid search nejen pro své interní hyperparametry, ale také v rámci výběru příznaků (feature selection), aby bylo dosaženo optimální konfigurace pro každý přístup. Každý regresní model byl laděn nezávisle a nejlépe fungující regresní model byl poté zvolen jako základní srovnávací referenční bod vůči nejlépe fungující neuronové síti, přičemž oba modely byly hodnoceny za shodných podmínek.

Modely byly testovány na reálném datasetu Use Case Points (UCP) obsahujícím 70 softwarových projektů. Pomocí trojnásobné křížové validace (3-fold cross-validation) byla predikční výkonnost měřena pomocí koeficientu determinace (R^2) a střední kvadratické chyby (RMSE).

Výsledky ukazují, že kompaktní neuronová síť se čtyřmi neurony ve skryté vrstvě překonala všechny regresní modely jak z hlediska přesnosti, tak i schopnosti generalizace. Zatímco regresní modely nabídly interpretovatelné výchozí hodnoty, vykazovaly vyšší rozptyl chyb a menší robustnost při zachycování složitějších vzorců. Naše zjištění jsou v souladu s předchozími studiemi, které naznačují, že neuronové sítě – pokud jsou správně nakonfigurovány – mohou překonat tradiční regresní techniky i v podmínkách s omezeným množstvím dat.

Klíčová slova: Software Effort Estimation, Hluboké neuronové sítě, Regresní modely, Use Case Points, Grid Search, Feature Selection, Křížová validace, Střední kvadratická chyba, Koeficient determinace, Porovnání modelů

ABSTRACT

This thesis explores the effectiveness of deep neural networks in comparison to traditional regression models for software effort estimation. Following a literature review of predictive techniques and evaluation metrics, a structured experiment was designed to compare four regression models—Linear Regression, Polynomial Regression, Lasso, and Ridge—with a feedforward neural network.

Regression models underwent grid search not only for their internal hyperparameters but also for feature selection, ensuring optimal configuration for each approach. Each regression model was tuned independently, and the best-performing regression model was selected as a baseline for comparison with the best-performing neural network, both evaluated under the same conditions.

The models were tested on a real-world Use Case Points (UCP) dataset containing 70 software projects. Using 3-fold cross-validation, predictive performance was measured with the coefficient of determination (R^2) and Root Mean Squared Error (RMSE).

The results show that a compact neural network using just 4 neurons in its hidden layer outperformed all regression models in both accuracy and generalization. While regression models offered interpretable baselines, they exhibited greater error variance and were less robust in capturing complex patterns. Our findings are consistent with prior studies suggesting that neural networks, when properly configured, can outperform traditional regression techniques even in low-data scenarios.

Keywords: Software Effort Estimation, Deep Neural Networks, Regression Models, Use Case Points, Grid Search, Feature Selection, Cross-Validation, RMSE, R^2 Score, Model Comparison

TABLE OF CONTENTS

1	INTRODUCTION	11
1.1	THE IMPORTANCE OF SOFTWARE EFFORT ESTIMATION.....	12
1.2	OVERVIEW OF EXISTING METHODS FOR SOFTWARE EFFORT ESTIMATION	12
1.2.1	Expert Judgment	13
1.2.2	Analogous Estimation.....	13
1.2.3	Parametric Models	13
1.2.4	Machine Learning Techniques	14
1.2.5	Function Point Analysis (FPA).....	14
1.2.6	Use Case Points (UCP)	14
1.2.7	Story Points and Agile Estimation	14
1.3	TRADITIONAL METHODS.....	15
1.3.1	COCOMO	15
1.3.2	Function Points	17
1.3.3	Regression Models.....	18
1.4	MACHINE LEARNING.....	22
1.4.1	Random Forest.....	24
1.4.2	Support Vector Machines.....	25
1.4.3	Neural Networks	27
1.5	DEEP NEURAL NETWORKS	30
1.6	COMPARISON OF TRADITIONAL AND MODERN METHODS	31
1.7	CHALLENGES AND FUTURE DIRECTIONS	33
2	METHODOLOGY.....	36
2.1	DATASET OVERVIEW.....	36
2.2	TARGET VARIABLE: REAL EFFORT PERSON HOURS.....	39
2.2.1	Implications for Modeling	41
2.3	EXPLORATORY DATA ANALYSIS	42
2.4	FEATURE DESCRIPTION.....	47
2.5	FEATURE SELECTION.....	52

2.6	DATA PREPROCESSING	54
2.6.1	Data Preprocessing implementation	56
2.7	EVALUATION METRICS AND VALIDATION STRATEGY	59
2.7.1	Performance Metrics	59
2.7.2	Validation Methods	61
2.8	IMPLEMENTATION	63
2.8.1	Software and Libraries Used	63
2.8.2	Regression Models Implementation	65
2.8.3	Deep Neural Networks Implementation.....	74
3	RESULTS.....	82
3.1	VISUAL COMPARISON OF PERFORMANCE	83
4	CONCLUSION AND FUTURE WORK.....	89
	REFERENCES	91
	LIST OF FIGURES	93
	LIST OF TABLES	94

1 INTRODUCTION

In this work, we compare the performance of traditional regression models and deep neural networks in the task of software effort estimation. The main objective is to determine which modeling approach is more suitable for predicting software development effort based on structured input data. To achieve this, we design an experiment that evaluates four regression models (Linear, Polynomial, Ridge, and Lasso) and compares them to a feedforward neural network using a common dataset and consistent validation methodology.

The thesis is structured into four main parts. The **Introduction** provides context for the problem of software effort estimation, reviews relevant literature, and briefly explains how regression models and neural networks work and why they are used in this domain. The **Methodology** section describes the dataset used in this study, explains the meaning of its features within the Use Case Points (UCP) framework, details the preprocessing steps, and outlines the experimental setup including model evaluation metrics and implementation details. The **Results** section presents and compares the performance of all tested models, highlighting the differences in accuracy and generalization. Finally, the **CONCLUSION AND FUTURE WORK** summarizes key findings, discusses the implications of the results, and outlines possible directions for future research.

1.1 The importance of software effort estimation

Software effort estimation plays a crucial role in software engineering, directly influencing a project's success, budget, and timeline. As Steve McConnell points out in *Software Estimation: Demystifying the Black Art*, estimation isn't just about crunching numbers—it's a strategic tool that helps teams plan effectively and avoid costly mistakes. Without accurate estimates, projects can run over budget, miss deadlines, or even fail entirely, wasting resources and damaging reputations.

Early on, estimating software effort was more of an educated guess than a structured process. Developers relied on personal experience, using expert judgment and analogy-based methods, but these approaches often led to inconsistent results. The introduction of algorithmic models, such as Barry Boehm's COCOMO in the 1980s, brought a shift toward data-driven estimation, leveraging historical project data to improve accuracy. Even with these advancements, predicting software effort remains a difficult task because of the unique challenges and unpredictability of each project.

McConnell emphasizes that estimation is not about making perfect predictions—it's about managing uncertainty and making informed decisions. As software continues to grow in complexity, effort estimation remains essential for keeping development on track, optimizing resources, and ensuring projects align with business goals [1].

1.2 Overview of Existing Methods for Software Effort Estimation

Software effort estimation is a key part of project management, helping teams predict how much work will be needed to develop a software system. Accurate estimates are crucial for planning, budgeting, and allocating resources efficiently. Over the years, various estimation methods have been developed, each with its own strengths and best-use scenarios depending on the nature of the project. Below is an overview of some of the most commonly used techniques.

1.2.1 Expert Judgment

One of the simplest and most widely used methods, expert judgment relies on the knowledge and experience of seasoned professionals to estimate effort based on past projects. This approach is highly flexible and can be adapted to different project types. However, because it is based on individual expertise, estimates may vary between experts and can sometimes be subjective [2].

1.2.2 Analogous Estimation

Also known as comparative estimation, this method involves looking at previous projects with similar characteristics to predict effort for a new one. By leveraging historical data, estimators can make informed guesses about development time and resource requirements. While this approach can provide reasonable estimates, its accuracy depends on the availability and quality of past project data [2].

1.2.3 Parametric Models

Parametric models take a more structured approach by using mathematical formulas to establish relationships between software metrics (such as lines of code or function points) and effort. Some of the most recognized parametric models include:

- **COCOMO (Constructive Cost Model):** Developed by Barry Boehm, COCOMO estimates development effort based on software size and various cost drivers, such as project complexity and team capability. This model has been widely adopted and refined over time to improve accuracy [3].
- **Putnam Model (SLIM):** Proposed by Lawrence H. Putnam, SLIM is based on empirical data and describes the relationship between effort, time, and project size. It is particularly useful for long-term planning and assessing the feasibility of large-scale projects [2].

1.2.4 Machine Learning Techniques

With advancements in data science, machine learning algorithms have been applied to effort estimation. These techniques learn patterns from historical project data to predict effort for new projects. Methods such as regression analysis, neural networks, and decision trees are commonly employed [4].

1.2.5 Function Point Analysis (FPA)

FPA measures software size by quantifying the functionality provided to the user, independent of the technology used for implementation. This size metric is then used to estimate the effort required, considering factors like complexity and performance requirements [2].

1.2.6 Use Case Points (UCP)

Estimates from the UCP method were close to those made by experienced developers, with acceptable error rates [5]. This method estimates effort based on the number and complexity of use cases in the system. By evaluating actors and transactions within use cases, estimators can derive an effort approximation [2, 6].

1.2.7 Story Points and Agile Estimation

Commonly used in Agile methodologies, story points represent the relative effort of implementing user stories. Teams assign point values based on complexity, risk, and size, facilitating iterative planning and estimation. Teams typically employ Planning Poker, an agile estimation technique where team members assign story point values (using a Fibonacci-like sequence 1, 2, 3, 5, 8, ...) to each story and reach a consensus [7].

Each of these methods has its advantages and limitations. The choice of estimation technique should align with the project's characteristics, available data, and organizational context to ensure accuracy and reliability in effort predictions [2, 8, 4].

1.3 Traditional Methods

1.3.1 COCOMO

COCOMO Methodology The Constructive Cost Model (COCOMO), developed by Barry W. Boehm, is one of the most widely recognized and applied models for software effort estimation. It provides a structured, algorithmic approach to predicting software development effort, cost, and schedule based on project size and a set of influencing factors. By leveraging historical project data and well-defined estimation equations, COCOMO enhances the predictability and control of software projects [3].

COCOMO exists in multiple forms, evolving over time to accommodate the increasing complexity and variability of software development practices. The original model, COCOMO 81, consists of three hierarchical estimation models:

- **Basic COCOMO:** The simplest form, this model estimates effort purely based on the number of delivered source instructions (KDSI). It classifies projects into three categories:
 - *Organic*: Small, relatively simple projects with experienced teams and flexible requirements.
 - *Semi-Detached*: Intermediate-sized projects with mixed experience levels and some system constraints.
 - *Embedded*: Complex, tightly constrained projects with strict operational requirements.

Each project type has its own predefined effort equation, but the model lacks sensitivity to factors such as personnel capability, tools, and methodologies

- **Intermediate COCOMO:** Extending the Basic model, Intermediate COCOMO incorporates 15 cost drivers that adjust the effort estimation based on attributes such as software complexity, development environment, personnel experience, and required reliability. This provides a more realistic assessment of effort and duration [3].

- **Detailed COCOMO:** This model further refines effort estimation by considering the effort distribution across different phases of development (e.g., requirements analysis, design, coding, testing). It enables project managers to allocate resources more precisely throughout the software lifecycle [9].

COCOMO II: Advancements and Modernization With the emergence of new development paradigms, increased software reuse, and changing project structures, the original COCOMO model was refined into COCOMO II. This newer version accounts for modern software engineering practices, such as component-based development, incremental and iterative lifecycles, and the use of commercial off-the-shelf (COTS) components [3].

COCOMO II consists of three sub-models, each tailored to different stages of the software development lifecycle:

- **Application Composition Model:** Used for early effort estimation in projects where rapid development environments (such as Visual Basic or Python frameworks) are utilized. This model measures effort based on object points, which quantify user interface complexity and integration effort [3].
- **Early Design Model:** Designed for the early estimation of new software projects when only high-level requirements are available. Instead of relying on detailed size measurements, it utilizes function points and high-level cost drivers [3].
- **Post-Architecture Model:** The most detailed and precise model in COCOMO II, this approach is used when the project architecture is established. It considers detailed cost drivers, adjusted effort multipliers, and software size metrics such as function points or source lines of code (SLOC) [3].

Mathematical Representation of COCOMO II COCOMO II expresses effort estimation through an empirical formula:

$$E = A \times \text{Size}^B \times \prod_{i=1}^n EM_i$$

where:

- E is the estimated effort (measured in person-months).
- A is a calibration constant derived from historical project data.
- Size represents the software size (in KDSI or function points).
- B is the scale exponent reflecting economies or diseconomies of scale.
- EM_i are effort multipliers corresponding to cost drivers such as required software reliability, product complexity, and developer experience [3].

Significance and Impact of COCOMO COCOMO remains one of the most influential models in software engineering due to its structured approach and adaptability. It provides a balance between simplicity and accuracy, making it suitable for both early and detailed effort estimations. Organizations use COCOMO not only for cost prediction but also for resource allocation, risk assessment, and process optimization. While newer estimation techniques, such as machine learning models, offer alternative approaches, COCOMO continues to serve as a fundamental framework for understanding software cost estimation [3].

1.3.2 Function Points

Function Point Analysis (FPA) is a widely used method for measuring the functional size of a software system, providing an objective metric for software development effort estimation, productivity assessment, and project management. Unlike traditional size measurement techniques such as Lines of Code (LOC), which are technology-dependent, function points evaluate software based on the functionalities delivered to the end user, making it a technology-agnostic approach [10].

The function point methodology classifies system functions into five main categories:

- **External Inputs (EI):** User-originated inputs that alter system data or control information, such as data entry forms or API requests.

- **External Outputs (EO):** Information sent from the system to the user, including reports, messages, or displayed results.
- **External Inquiries (EQ):** Interactive inputs that require system responses but do not modify internal data, such as search queries or system-generated prompts.
- **Internal Logical Files (ILF):** Persistent data maintained within the system for processing needs, such as a customer database.
- **External Interface Files (EIF):** Data that the system references but does not maintain, such as linked external databases or shared files [10].

Each function is assigned a weight (low, average, or high) based on its complexity, and the total function point count is determined using a predefined formula. Additionally, FPA incorporates 14 general system characteristics, such as transaction rates, performance requirements, and data processing complexity, which further refine the final function point count [10].

Function points serve as a foundation for various project management activities, including software effort estimation, benchmarking, and productivity measurement. They enable organizations to compare project sizes across different technologies and development environments, providing a standardized basis for performance evaluation. Furthermore, function point analysis aids in cost estimation, facilitating better budgeting and resource allocation by predicting the effort required for software development and maintenance tasks [10].

Due to its ability to provide a consistent, technology-independent measure of software functionality, function point analysis has been widely adopted in industries that require precise software effort estimation. It remains an essential tool for project managers and organizations seeking to improve software development predictability and efficiency [10].

1.3.3 Regression Models

Regression models play a significant role in software effort estimation by establishing relationships between project characteristics and effort predictions. These models leverage

historical project data to derive mathematical equations that estimate the required development effort based on measurable attributes such as software size, complexity, team experience, and technology factors. The primary advantage of regression-based estimation is its ability to provide objective, data-driven predictions while reducing subjective bias in effort assessment [2].

Understanding Regression for Software Estimation Regression analysis is a statistical technique used to model the relationship between a dependent variable (software development effort) and one or more independent variables (such as project size, team experience, or complexity). The general form of a regression model can be expressed as:

$$E = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n + \varepsilon$$

where:

- E is the estimated software development effort (in person-months or other relevant units).
- β_0 is the intercept (baseline effort when all independent variables are zero).
- β_i are the regression coefficients representing the impact of each independent variable X_i .
- X_i are the predictor variables (e.g., software size, complexity).
- ε is the error term accounting for variability not captured by the model [2].

Regression models are particularly useful in software estimation due to their ability to analyze large datasets, recognize patterns, and adapt to different project contexts. Depending on the complexity of the relationships among project factors, different types of regression models are applied.

Types of Regression Models in Software Estimation Several types of regression models are employed in software effort estimation, each with its own strengths and limitations:

Simple Linear Regression Simple linear regression models effort as a function of a single independent variable, typically software size (e.g., lines of code or function points). The model takes the form:

$$E = \beta_0 + \beta_1 X + \varepsilon$$

where X represents the software size metric. While easy to interpret, this model oversimplifies effort estimation by ignoring other influencing factors such as team expertise, development environment, and system complexity [2].

Multiple Linear Regression To address the limitations of simple regression, multiple linear regression includes multiple independent variables:

$$E = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n + \varepsilon$$

This approach allows effort estimation to incorporate various project attributes, such as team size, project duration, and code reuse, improving prediction accuracy. However, it assumes linear relationships between effort and predictors, which may not always hold in complex software projects [2].

Polynomial Regression Polynomial regression extends multiple linear regression by introducing nonlinear relationships between variables. The model includes higher-degree terms:

$$E = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_n X^n + \varepsilon$$

This allows for more flexible modeling of software effort trends, particularly when effort exhibits nonlinear dependencies on size or complexity. However, overfitting can become a

concern if the model is too complex relative to the dataset size [2].

Logarithmic and Exponential Regression Some software estimation models utilize logarithmic or exponential transformations to capture nonlinear relationships. Examples include:

$$E = \beta_0 + \beta_1 \log(X) + \varepsilon$$

$$E = \beta_0 e^{\beta_1 X} + \varepsilon$$

These models are particularly useful when effort grows exponentially with project size or complexity, as seen in large-scale software systems with high interdependencies [2].

Stepwise Regression Stepwise regression is a data-driven approach that iteratively selects the most significant predictors for effort estimation. This technique helps in avoiding multicollinearity (correlation between independent variables) and reduces the risk of overfitting by including only relevant features in the model. It is commonly used in empirical software engineering studies to refine regression models [2].

Advantages and Challenges of Regression Models in Software Estimation Regression models provide a robust statistical framework for effort estimation by offering interpretable relationships between project characteristics and development effort. They facilitate early-stage planning, resource allocation, and risk assessment while leveraging historical data for predictive accuracy. However, their effectiveness depends on the quality of input data, the selection of appropriate predictors, and the assumption that past project trends are representative of future developments [2].

Furthermore, regression models assume a fixed relationship between variables, which may not fully capture the dynamic nature of software projects influenced by evolving methodologies, team skill variations, and changing technologies. Hybrid approaches, combining regression with machine learning techniques, have emerged to enhance estimation accuracy by adapting to complex, nonlinear dependencies within software project data [2].

Overall, regression models remain a fundamental tool in software effort estimation, offering a structured, statistical approach to quantifying development effort while providing insights into key cost drivers and project complexities.

1.4 Machine Learning

Machine learning (ML) has emerged as a powerful alternative to traditional software effort estimation methods by leveraging data-driven techniques to identify complex patterns in historical project data. Unlike regression-based models that rely on predefined mathematical relationships, ML algorithms automatically learn patterns and dependencies from training data, making them particularly effective in modeling nonlinear relationships and high-dimensional datasets [11, 12].

Application of Machine Learning in Effort Estimation ML techniques provide a flexible and scalable approach to software effort estimation, particularly when dealing with complex, noisy, or incomplete datasets. These methods do not rely on strong parametric assumptions and instead learn from data distributions through optimization and iterative refinement. Commonly used ML models in this domain include:

- **Decision Trees and Random Forests:** These non-parametric models partition the input space into regions based on learned decision rules. While individual decision trees are easy to interpret but prone to overfitting, random forests mitigate this by aggregating predictions from multiple trees, thereby improving generalization and reducing variance [4].
- **Artificial Neural Networks (ANNs):** Including both shallow and deep architectures, ANNs capture hierarchical feature interactions through multiple layers of interconnected nodes. They have demonstrated strong predictive performance in effort estimation tasks, particularly when trained on sufficiently large and well-engineered datasets [13, 14].
- **Support Vector Machines (SVMs):** Originally developed for classification, SVMs are also used in regression tasks (SVR). By employing kernel functions, they effec-

tively handle nonlinear relationships and high-dimensional input spaces, making them suitable for complex effort estimation scenarios [15].

- **Ensemble and Hybrid Methods:** These methods combine multiple base learners—such as in boosting, bagging, or model stacking—to leverage the strengths of different algorithms. Ensemble approaches often outperform single models, particularly when individual learners exhibit complementary biases.

Unlike traditional statistical approaches, machine learning methods focus on minimizing predictive error through iterative optimization and model validation. They often require careful hyperparameter tuning, feature engineering, and robust evaluation techniques (e.g., cross-validation) to avoid overfitting and ensure reliable performance.

Advantages and Challenges Machine learning can be more effective than traditional methods [16]. Machine learning provides several advantages over traditional effort estimation models. It can dynamically adapt to evolving software development practices, leverage large and complex datasets, and improve predictive accuracy by capturing nonlinear relationships. Furthermore, ensemble techniques, such as boosting and bagging, reduce estimation errors by aggregating predictions from multiple models [12].

However, machine learning models also pose challenges. They require high-quality and sufficiently large datasets to generalize well, and their interpretability is often lower than traditional parametric models. Additionally, hyperparameter tuning and feature selection are crucial for achieving optimal performance, necessitating domain expertise and computational resources [15].

Conclusion Machine learning has transformed software effort estimation by introducing data-driven, adaptive models capable of capturing complex dependencies. As software projects continue to grow in complexity, integrating ML techniques with traditional estimation models offers a promising direction for improving the reliability and efficiency of software project management [11, 13].

1.4.1 Random Forest

Random Forests are an ensemble learning method that constructs multiple decision trees during training to enhance predictive accuracy and control overfitting. Each tree is built using a random subset of the training data and features, and the final prediction is obtained by aggregating the outputs of individual trees, typically through majority voting for classification tasks or averaging for regression tasks [17]. The general form of a decision tree model for effort estimation can be expressed as:

$$E = T(X)$$

where:

- E represents the estimated effort.
- X denotes the input feature vector, such as counts of use cases, actors, or complexity metrics.
- $T(X)$ is the decision function defined by the tree structure, which recursively partitions the input space based on feature thresholds.

Each internal node of the tree applies a decision rule (e.g., $X_i \leq \theta$) that splits the data into subsets, while leaf nodes assign a constant prediction value—typically the average effort of training samples that fall into that region. The tree is trained by minimizing a loss function, often the mean squared error (MSE), over all terminal nodes.

Decision trees are inherently interpretable, as each prediction can be traced through a series of if-then rules. Their non-parametric nature makes them well-suited for capturing non-linear relationships and feature interactions without requiring data normalization or transformation. However, without pruning or regularization, decision trees are prone to overfitting, especially in small datasets.

In the context of software effort estimation, Random Forests offer several advantages:

- **Handling High-Dimensional Data:** They can process datasets with a large number of predictor variables, such as project size metrics, team experience levels, and technological factors, without the need for extensive feature selection.
- **Capturing Nonlinear Relationships:** By combining multiple decision trees, Random Forests can model complex, nonlinear interactions between input variables and the target effort estimation, leading to more accurate predictions.
- **Robustness to Overfitting:** The ensemble nature of Random Forests reduces the risk of overfitting, which is a common issue in individual decision trees, especially when dealing with noisy data [17].

However, practitioners should be aware of certain limitations:

- **Interpretability:** While individual decision trees are easily interpretable, the aggregated model in a Random Forest is more complex, making it challenging to extract simple decision rules.
- **Computational Resources:** Training a large number of trees can be computationally intensive, which may be a consideration for very large datasets or real-time estimation requirements.

Despite these challenges, Random Forests have been successfully applied in software effort estimation, offering a balance between model complexity and predictive performance. Their ability to handle diverse data types and capture intricate patterns makes them a valuable tool in the software project management toolkit.

1.4.2 Support Vector Machines

Support Vector Machines (SVMs) are a powerful supervised learning algorithm widely used in classification and regression tasks, including software effort estimation. SVMs operate by identifying an optimal hyperplane that best separates data points in a high-dimensional

space, minimizing error while maximizing margin. For regression-based tasks, such as predicting software development effort, a variation known as Support Vector Regression (SVR) is employed [18].

How SVMs Work in Effort Estimation In software effort estimation, SVMs model the relationship between various project characteristics (e.g., lines of code, function points, team experience) and the estimated effort required for development. Unlike traditional linear regression models, SVMs use kernel functions to map input variables into a higher-dimensional space, enabling them to capture complex, nonlinear relationships. The general form of SVR is given by:

$$E = w \cdot \phi(X) + b$$

where:

- E represents the estimated effort.
- X denotes the input features, such as software size and complexity metrics.
- $\phi(X)$ is a transformation function that maps the input into a higher-dimensional space.
- w and b are model parameters optimized during training [18].

The training process involves minimizing the error within a specified margin ε , allowing the model to ignore small deviations while focusing on significant variations. This robustness makes SVR effective for handling noisy and complex datasets common in software estimation.

Advantages of SVMs in Software Estimation SVMs offer several advantages for software effort estimation:

- **Handling Nonlinear Relationships:** The use of kernel functions (e.g., radial basis

function, polynomial) allows SVMs to model intricate dependencies between effort and project attributes.

- **Resilience to Overfitting:** By maximizing margin and using regularization techniques, SVMs generalize well even when dealing with limited training data.
- **Effective with High-Dimensional Data:** SVMs perform well in scenarios where the number of features is large relative to the number of available training examples [18].

Challenges and Considerations Despite their advantages, SVMs also present challenges:

- **Computational Complexity:** Training SVMs can be computationally expensive, particularly with large datasets, as the optimization problem grows quadratically with the number of training samples.
- **Hyperparameter Tuning:** The performance of SVMs heavily depends on selecting appropriate kernel functions and hyperparameters, requiring cross-validation and domain expertise.
- **Interpretability:** Unlike linear regression models, SVMs are less interpretable, making it difficult to extract simple decision rules for effort estimation [18].

Conclusion SVMs provide a robust, data-driven approach to software effort estimation, offering superior predictive performance for complex and nonlinear datasets. While they require careful parameter tuning and computational resources, their ability to generalize well across various software projects makes them a valuable tool in modern software estimation methodologies [18].

1.4.3 Neural Networks

Neural networks (NNs) have emerged as a powerful machine learning technique for software effort estimation, offering the ability to model complex, nonlinear relationships between software project attributes and development effort. Artificial neural network models are

among the most frequently studied techniques in effort estimation research [16]. Unlike traditional regression models, which assume linear relationships, neural networks adaptively learn patterns from historical project data, making them particularly effective in handling noisy and high-dimensional datasets [14].

Structure and Functioning of Neural Networks A neural network consists of multiple interconnected layers of neurons, each performing mathematical transformations on input data. The most common type of NN used for effort estimation is the *multilayer perceptron* (MLP), which typically consists of:

- **Input Layer:** Receives project attributes such as size metrics (e.g., function points, lines of code), team experience, and development environment.
- **Hidden Layers:** Intermediate layers where weighted connections and activation functions (e.g., ReLU, sigmoid) help capture nonlinear dependencies between input variables.
- **Output Layer:** Produces the estimated software effort, typically in person-months or similar units [14].

The network is trained using optimization algorithms such as backpropagation and gradient descent, adjusting weights based on error minimization. The loss function, such as Mean Squared Error (MSE), guides the learning process to improve estimation accuracy.

Advantages of Neural Networks in Software Estimation Neural networks offer several advantages over traditional estimation techniques:

- **Ability to Model Nonlinearity:** Software effort is influenced by multiple interdependent factors. NNs automatically learn these relationships without requiring explicit feature engineering.
- **Adaptability to Large and Complex Datasets:** Unlike parametric models that may struggle with high-dimensional data, neural networks effectively scale with large datasets.

- **Feature Importance and Interaction:** By assigning different weights to input variables, neural networks naturally highlight influential factors in software estimation [14].

Challenges of Using Neural Networks Despite their advantages, neural networks also pose challenges in software effort estimation:

- **Data Dependency:** Large labeled datasets are required for training, but software estimation datasets are often limited in size.
- **Computational Complexity:** Neural networks require significant computational resources for training, particularly deep architectures with multiple hidden layers.
- **Hyperparameter Sensitivity:** The performance of NNs depends on selecting the right network architecture, learning rate, number of hidden layers, and activation functions, requiring extensive experimentation [14].

Hybrid Approaches and Future Trends To address these challenges, researchers have proposed hybrid approaches combining neural networks with traditional estimation models. For instance, integrating neural networks with statistical regression models can improve interpretability while retaining the predictive power of deep learning. Additionally, recent advancements such as recurrent neural networks (RNNs) and transformer-based models are being explored for sequential software project estimation tasks [14].

Conclusion Neural networks provide a robust and adaptive approach to software effort estimation, enabling organizations to improve prediction accuracy and decision-making. While they require careful tuning and large datasets, their ability to uncover complex patterns makes them a valuable tool in modern software engineering [14].

1.5 Deep neural networks

Deep neural networks (DNNs) represent an advanced class of neural architectures that extend traditional (shallow) networks by incorporating multiple hidden layers. While shallow networks typically contain one or two hidden layers, DNNs may include dozens or even hundreds of layers, enabling them to model highly nonlinear and hierarchical relationships in data. This depth allows DNNs to learn abstract and multi-level feature representations that are often inaccessible to shallow models [13].

Differences Between Shallow and Deep Learning The key distinction between shallow and deep learning lies in the model depth and representational capacity. Shallow networks are limited in their ability to capture complex functions, making them less effective for large-scale or high-dimensional tasks. In contrast, DNNs can automatically extract layered patterns from data—for example, low-level features, mid-level structures, and high-level concepts—making them well suited for problems with intricate data structures.

Variants of Deep Neural Networks Several specialized DNN architectures have been developed, each tailored to different data types and learning objectives:

- **Feedforward Deep Networks (DNNs):** The most basic deep architecture, where data flows unidirectionally through layers without recurrence. Commonly used for regression and classification on structured data.
- **Convolutional Neural Networks (CNNs):** Designed to process spatially correlated data such as images. Although less common in effort estimation, CNNs are effective for tasks involving graphical representations or temporal patterns with local dependencies.
- **Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM):** Suited for sequential data, these networks maintain internal memory to capture temporal dependencies. They can be applied to model time-series aspects of project effort or activity sequences.

- **Residual Networks (ResNets) and Transformer-based Architectures:** Used in very deep models, these networks solve issues such as vanishing gradients through mechanisms like skip connections (ResNets) or attention mechanisms (Transformers), enabling effective training of highly expressive architectures.

Optimization Algorithms for Training DNNs Training deep networks is computationally intensive and requires advanced optimization techniques to ensure convergence and model stability. While traditional gradient descent can be used, it is often replaced by more adaptive variants:

- **Stochastic Gradient Descent (SGD):** A foundational algorithm that updates weights based on mini-batches of data. Typically combined with momentum to smooth updates and escape shallow minima.
- **Adam (Adaptive Moment Estimation):** One of the most widely used optimizers, Adam computes individual adaptive learning rates for each parameter by combining momentum and RMSProp strategies. It performs well across a wide range of deep learning tasks with minimal tuning.
- **Other Methods:** Alternatives like RMSProp, Adagrad, and Nadam are also employed depending on the architecture and data characteristics. In practice, optimizer selection can significantly affect training dynamics and final model performance.

1.6 Comparison of Traditional and Modern Methods

Software effort estimation has evolved from rule-based, algorithmic models to data-driven machine learning techniques. Traditional methods, such as the Constructive Cost Model (COCOMO) and Function Point Analysis (FPA), rely on predefined mathematical relationships and expert-driven assessments. These approaches provide structured, interpretable models but often struggle with accuracy when applied to modern, complex software projects. In contrast, machine learning techniques offer adaptive, pattern-recognizing models that improve estimation accuracy by leveraging large-scale project data [19].

Traditional Estimation Methods Traditional software effort estimation models, including COCOMO, Putnam's SLIM, and regression-based models, use predefined formulas to predict effort based on input features such as software size (e.g., Lines of Code or Function Points), project complexity, and team experience. These models are deterministic, meaning they produce the same result given the same input, making them transparent and easy to audit. However, their key limitations include:

- **Dependence on Expert-Defined Parameters:** COCOMO and other traditional models require manual calibration of parameters, which may introduce biases or inaccuracies.
- **Difficulty in Handling Nonlinear Relationships:** Real-world software development effort is influenced by numerous nonlinear and interdependent factors that traditional models often fail to capture.
- **Limited Adaptability:** These models assume static relationships between effort predictors and do not learn from new project data over time [19].

Machine Learning-Based Estimation Methods Modern machine learning models, such as Artificial Neural Networks (ANNs), Random Forests, and Support Vector Machines (SVMs), learn from historical project data to predict effort more accurately. These techniques have several advantages over traditional methods:

- **Ability to Model Complex, Nonlinear Relationships:** Machine learning algorithms do not require explicit predefined equations; instead, they infer patterns and dependencies directly from the data.
- **Higher Prediction Accuracy:** Studies comparing traditional models with ML-based models show that machine learning generally outperforms rule-based estimation techniques, especially for heterogeneous and large datasets.
- **Automatic Feature Selection:** Algorithms like Random Forests can identify the most significant effort predictors dynamically, reducing the need for expert-driven feature selection [19].

Challenges of Machine Learning Approaches Despite their advantages, machine learning models also pose challenges:

- **Data Dependency:** ML models require large and high-quality datasets for training. Insufficient or biased training data can lead to inaccurate predictions.
- **Computational Cost:** Training complex models, such as deep neural networks, requires significant computational power and tuning of hyperparameters.
- **Lack of Interpretability:** Unlike traditional models with well-defined equations, some machine learning techniques, particularly deep learning models, act as "black boxes," making it difficult to explain their predictions in human-understandable terms [19].

Conclusion The shift from traditional to machine learning-based estimation methods represents a significant advancement in software effort estimation. While traditional models remain useful for their interpretability and ease of use, machine learning models provide superior accuracy and adaptability. Future research is expected to focus on hybrid approaches, combining the strengths of both methodologies to enhance prediction reliability and practical usability in diverse software development environments [19].

1.7 Challenges and Future Directions

Software effort estimation remains a complex task due to the inherent variability in software projects, changing development methodologies, and the limitations of existing models. While traditional models such as COCOMO and Function Point Analysis provide structured approaches to estimation, they often struggle with adaptability to modern development paradigms. On the other hand, machine learning-based models, including Support Vector Machines (SVMs), Random Forests, and Neural Networks, offer improved accuracy but introduce new challenges related to data requirements, interpretability, and computational complexity [1, 2].

Challenges in Software Effort Estimation Despite advancements in estimation methodologies, several challenges persist:

- **Data Availability and Quality:** Machine learning-based approaches require large, high-quality datasets for training. However, effort estimation datasets are often small, noisy, or biased, leading to suboptimal model performance [11, 12].
- **Feature Selection and Engineering:** Identifying the most relevant predictors of software effort is critical for accurate estimation. Traditional models use predefined features, while machine learning approaches must learn important features from the data, often requiring expert intervention [4].
- **Interpretability and Transparency:** Traditional models provide clear estimation equations, making them easy to interpret. In contrast, complex models such as Neural Networks and ensemble methods like Random Forests often function as "black boxes," making it difficult to understand how predictions are made [15, 17].
- **Computational Requirements:** Modern estimation models, particularly deep learning-based approaches, require significant computational resources for training and optimization, making them impractical for certain applications [13].
- **Generalization Across Projects:** Software projects vary widely in terms of technology, team experience, and requirements. Estimation models trained on one dataset may not generalize well to new projects, requiring continuous adaptation and retraining [14, 18].

Future Directions in Software Effort Estimation To address these challenges, ongoing research is focused on the following areas:

- **Hybrid Estimation Models:** Combining traditional estimation techniques with machine learning methods can enhance accuracy while maintaining interpretability. For example, hybrid models integrating COCOMO with machine learning algorithms could balance explainability with predictive power [3, 2].

- **Automated Feature Engineering:** Advances in deep learning and automated machine learning (AutoML) are expected to improve feature selection and representation learning, reducing the dependency on domain expertise [13, 12].
- **Explainable AI (XAI) for Effort Estimation:** Developing interpretable machine learning models will help stakeholders trust and understand effort predictions. Techniques such as SHAP (Shapley Additive Explanations) and LIME (Local Interpretable Model-agnostic Explanations) are promising directions for enhancing model transparency [11, 15].
- **Transfer Learning and Domain Adaptation:** Using pre-trained models and transfer learning approaches can help address the challenge of small datasets, enabling models trained on one software domain to be adapted for other domains with minimal data [14].
- **Integration with Agile and DevOps:** Modern software development practices emphasize continuous integration and delivery. Future effort estimation models must adapt to dynamic requirements and real-time feedback loops, integrating with Agile methodologies and DevOps pipelines [19].

Conclusion The field of software effort estimation is evolving rapidly, with machine learning playing an increasingly significant role. While traditional models remain relevant for their simplicity and transparency, modern approaches provide greater accuracy at the cost of interpretability and computational efficiency. Future research will likely focus on developing hybrid models that leverage the strengths of both methodologies, improving feature selection automation, and ensuring models remain adaptable to emerging software engineering practices [2, 11, 13].

In this work, we investigate whether a deep neural network can outperform traditional regression models in the task of software effort estimation, using a small dataset of 70 entries derived from the Use Case Points (UCP) methodology. Our primary research question is: Can a properly configured neural network achieve better predictive performance than regression models on a limited UCP-based dataset?

2 METHODOLOGY

2.1 Dataset Overview

For the comparison of deep neural networks and regression models in software effort estimation, a real-world dataset was utilized. The dataset consists of 70 software projects, each described by 39 features. These features consist of both individual metrics, as well as compound metrics as described by Software estimation literature.[3].

Compound features are:

- Unadjusted Actor Weight (UAW)
- Unadjusted Use Case Weight (UUCW)
- Technical Complexity Factors (TCF)
- Environmental Complexity Factors (ECF)

Target variable is:

- Real recorded effort

The dataset consists of:

- 70 rows (individual projects)
- 39 columns (features describing the projects)

Missing Values An analysis of missing values revealed that some features have incomplete data:

- **Sector:** 8 missing entries
- **Language:** 15 missing entries

- **Methodology:** 48 missing entries
- **ApplicationType:** 21 missing entries

Most technical and environmental features are complete with no missing values. There is a large number of missing entries in some categorical variables, particularly *Methodology*. We shall see in feature selection wheather we will find those variables usefull or not, if they are to be used they will need to be either imputed or entire row deleted. Loosing a row on such a small dataset is not favourable so we will try to avoid it if possible.

We have 5 categorical variables: Sector Language, Methodology, ApplicationType, DataDonator. Language refers to programming language used. Methodology refers to the development strategy used. Application type reffers to sector in which application is located, and finally the data donator refers to the source of individual data. Sector could potentially be a good predictor as individual sectors are well represented, See Table 2.1.

Insurance	12
Government	9
Banking	7
Communication	7
Service Industry	7
Manufacturing	6
Medical & Health Care	6
Wholesale & Retail	3
Electronics & Computers	3
Professional Services	2

Table 2.1 **Individual categories in sector are well represented**

For the chosen language, the dataset is dominated by the C# language, see Table 2.2.

Multiple languages are only present once, and are not going to be useful as predictors due to their low counts. For methodology used, Waterfall was used almost exclusively, therefore methodology is not a useful predictor, see Table 2.3

For application type the data is dominated by Business Application , see Table 2.4.

C#	25
Java	9
Visual Basic	6
C++	4
C	2
SQL	2
CSP	1
.Net	1
XML	1
Oracle	1
DELPHI	1
ASP	1
PowerBuilder	1

Table 2.2 The programming languages used in the projects and their counts. C# was the most frequently used language. The languages only used once or a handful of times are not useful as predictors.

Waterfall	15
Multifunctional Teams	2
Personal Software Process (PSP)	1
Rapid Application Development	1
Unified Process and its variants	1
Incremental	1
Rapid Application Development (RAD)	1

Table 2.3 The methodology used in the projects and their counts. Waterfall was almost exclusively the only methodology used. Therefore, this predictor is not useful

Business Application	40
Mathematically-Intensive Application	4
Real-Time Application	3
Real-Time application	1
Infrastructure Software	1

Table 2.4 The application types used in the projects and their counts. Business applications dominated the dataset, while other types were underrepresented.

Basic Statistics The basic descriptive statistics for the numerical features are summarized as follows:

- The **UAW** (Unadjusted Actor Weight) has a mean of 10.49, ranging from 6 to 19.
- The **UUCW** (Unadjusted Use Case Weight) has a mean of 386.2, with a minimum of 250 and a maximum of 610.
- **TCF** mean: 0.922, standard deviation: 0.115
- **ECF** mean: 0.836, standard deviation: 0.094

This initial analysis revealed that all values were within reasonable ranges, and we can thus proceed.

2.2 Target Variable: Real Effort Person Hours

In this study, the main target variable used for prediction is **Real Effort Person Hours**. This variable represents the actual amount of labor time, expressed in person-hours, that was consumed in completing a given software project. It is a direct quantitative indicator of the total effort invested in development, encompassing activities such as requirement analysis, design, coding, testing, and documentation.

From a software estimation perspective, Real Effort Person Hours serves as the most critical outcome metric. Accurate estimation of this value allows project managers to allocate resources effectively, plan timelines, and control budgets. Over- or underestimation can lead to project delays, cost overruns, or resource underutilization. Consequently, modeling this variable with high predictive accuracy is central to improving software project planning and management.

To better understand the behavior of the target variable, a histogram of Real Effort Person Hours was generated (see Figure 2.1). This visualization helps to assess the distribution of effort values across the dataset.

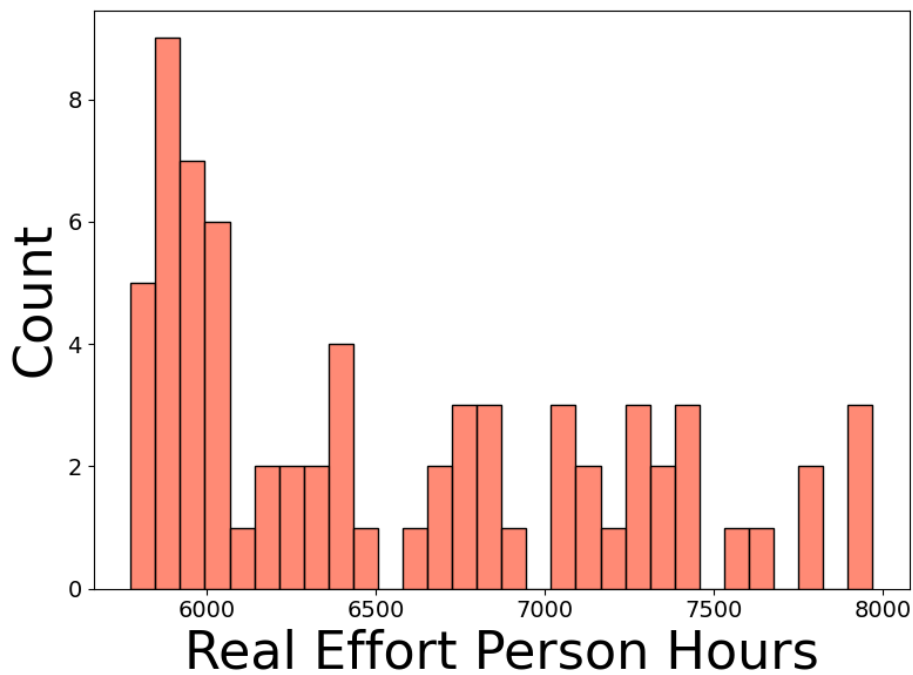


Figure 2.1 Distribution of the target variable: Real Effort Person Hours

The histogram reveals a multimodal distribution with a noticeable right skew. While a large portion of projects fall between approximately 5800 and 6300 person-hours, several distinct peaks extend toward higher values, reaching close to 8000 person-hours. This indicates the presence of project clusters with significantly different effort requirements, possibly corresponding to varying project types or data sources.

To further investigate the origin of these clusters, a boxplot was constructed to analyze the Real Effort Person Hours grouped by DataDonator (Figure 2.2).

The boxplot clearly shows that the three data contributors (D1, D2, D3) exhibit distinct effort distributions. Donator D1 provided higher-effort projects, while D3's entries are notably lower and more tightly clustered. This suggests that the source of the data plays a significant role in shaping the target distribution and explains the observed peaks in the histogram.

However, despite its correlation with effort values, DataDonator was not included as an input feature in the models. The reason is practical: this attribute does not represent an intrinsic characteristic of the software projects themselves, but rather a metadata artifact tied to the origin of the data. Including it could lead to information leakage or overly optimistic model performance that does not generalize to unseen donors or real-world scenarios.

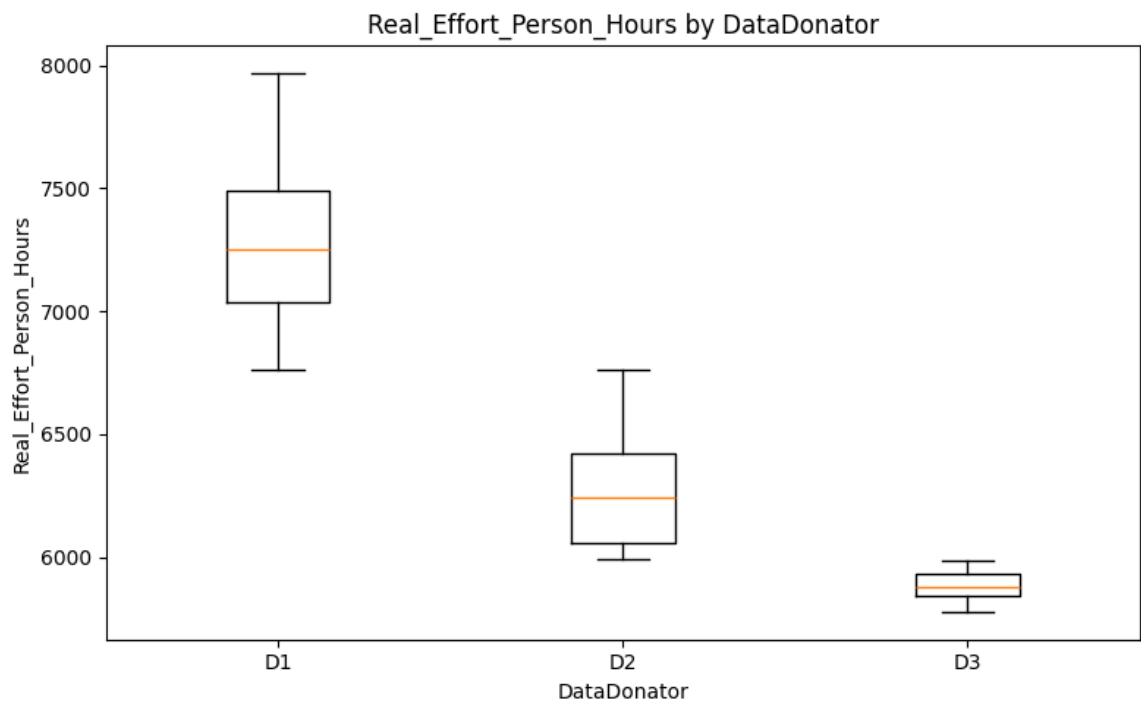


Figure 2.2 Boxplot of Real Effort Person Hours by DataDonator

In conclusion, while the clustering in effort can be traced back to data source variability, this factor is excluded from modeling to preserve the generalizability and fairness of the prediction models.

2.2.1 Implications for Modeling

Understanding the distribution of the target variable is essential for choosing suitable pre-processing techniques. The histogram reveals a moderate right skew, which could suggest applying a logarithmic transformation to stabilize variance—particularly in the context of linear regression. However, after evaluating both options, we decided to proceed with standardization using `StandardScaler` and omit logarithmic scaling.

This decision was made for several reasons. First, standardization brings the target variable to a zero-mean, unit-variance scale, which improves numerical stability and convergence in neural networks, especially when paired with a mean-squared error loss. Second, while log transformation can reduce skew, it also distorts the original scale of the prediction target, making interpretation less intuitive. Lastly, initial experiments showed that standardization alone provided satisfactory results, and the added complexity of log-based inverse transfor-

mation was not justified by significant performance gains.

In summary, the presence of right-skewed high-effort outliers was acknowledged, but standardization was deemed sufficient to accommodate them in both linear and neural models. More complex transformations were intentionally avoided to preserve interpretability and simplicity in the modeling pipeline.

2.3 Exploratory Data Analysis

Before building predictive models, exploratory data analysis (EDA) was performed to better understand the dataset's characteristics and prepare the data for modeling.

Correlation heatmap To better understand relationships in the data we will examine a correlation heatmap:

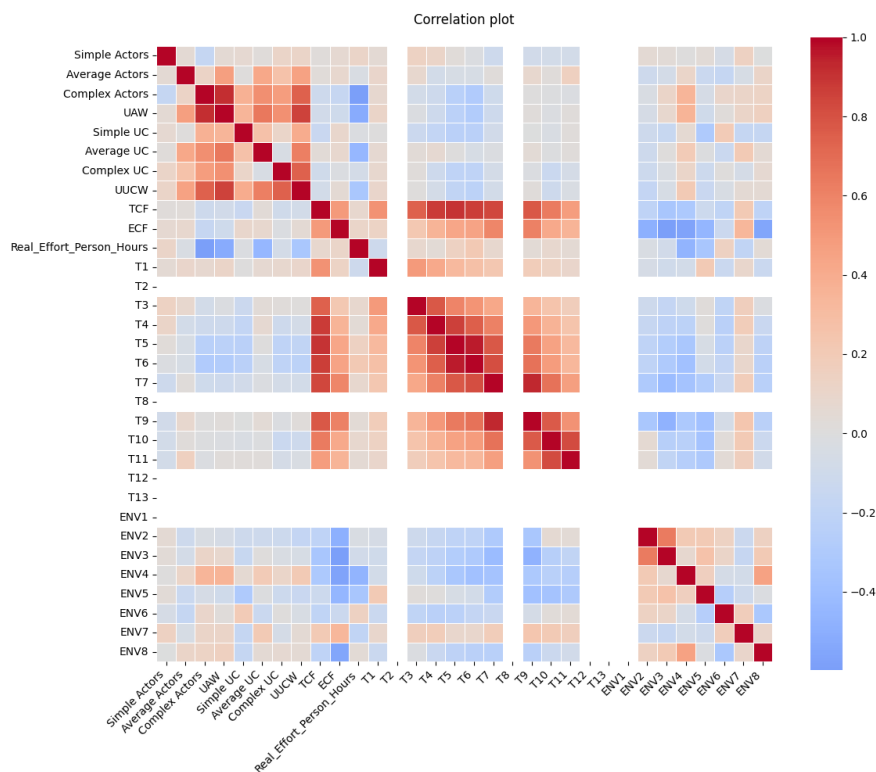


Figure 2.3 Correlation Heatmap of Variables

The correlation heatmap provides several meaningful insights into the relationships between the features in the dataset. Firstly, it is clearly observable that all the individual techni-

cal complexity factors (T1–T13) exhibit strong positive correlations with each other. This internal consistency indicates that the technical attributes of a project, such as distributed system requirements, performance objectives, or security concerns, tend to increase or decrease together. This is conceptually sound, as projects that demand, for example, high performance are also more likely to involve complex internal processing or concurrency requirements. The aggregated metric TCF (Technical Complexity Factor), which is computed as a weighted combination of these individual components, also correlates highly with its constituents. This reinforces the validity of the aggregation procedure and supports the notion that TCF can serve as a reliable summary indicator of technical project complexity.

Interestingly, an opposite trend is observed for the Environmental Complexity Factor (ECF) and its underlying components (ENV1–ENV8). Unlike TCF, ECF demonstrates a predominantly negative or weak correlation with its individual elements. This result is somewhat counterintuitive, as one would expect an aggregated index to maintain at least moderate correlation with its components. A potential explanation may lie in the weighting scheme or scoring inconsistencies used to derive ECF. Alternatively, the negative correlations could be an artifact of multicollinearity or compensatory scoring, where higher scores in one environmental factor are balanced by lower scores in another (e.g., high team motivation may offset the negative impact of unstable requirements). These findings suggest that the role of ECF in predictive modeling should be critically evaluated, and further investigation into its construction and empirical behavior may be warranted.

In summary, the correlation structure supports the use of technical factors and their aggregation in modeling effort, while raising questions about the empirical behavior of environmental factors. This insight could be valuable for feature selection, dimensionality reduction, and interpretation in both regression and machine learning models.

Distributions and Outliers To gain a deeper understanding of the dataset and identify any extreme values that could potentially distort the modeling process, an outlier analysis was conducted. Two standard statistical techniques were employed: the Interquartile Range (IQR) method and the Z-score method. The IQR-based analysis is summarized in Table 2.5, while the results obtained via Z-score analysis are presented in Table 2.6.

Table 2.5 Outlier Detection Results: IQR Method

Column	Outliers Count	Lower Bound	Upper Bound	Min Outlier	Max Outlier
Project_No	0	-32.5000	105.5000	NaN	NaN
Simple Actors	2	-1.5000	2.5000	4.00	4.00
Average Actors	0	-3.0000	5.0000	NaN	NaN
Complex Actors	0	-1.0000	7.0000	NaN	NaN
UAW	0	-9.1250	33.8750	NaN	NaN
Simple UC	2	-3.5000	8.5000	9.00	20.00
Average UC	2	4.0000	28.0000	3.00	30.00
Complex UC	0	0.1250	27.1250	NaN	NaN
UUCW	0	69.3750	724.3750	NaN	NaN
TCF	0	0.5950	1.2750	NaN	NaN
ECF	2	0.5500	1.1900	0.51	0.51
Real_P20	0	213.4625	439.4625	NaN	NaN
Real_Person_Hours	0	4269.2500	8789.2500	NaN	NaN
T1	10	0.0000	0.0000	1.00	5.00
T2	0	2.0000	2.0000	NaN	NaN
T3	0	-6.0000	10.0000	NaN	NaN
T4	0	-7.5000	12.5000	NaN	NaN
T5	0	-7.5000	12.5000	NaN	NaN
T6	0	-7.5000	12.5000	NaN	NaN
T7	16	2.5000	6.5000	0.00	0.00
T8	0	1.0000	1.0000	NaN	NaN
T9	16	5.0000	5.0000	0.00	4.00
T10	9	5.0000	5.0000	0.00	2.00
T11	5	5.0000	5.0000	3.00	4.00
T12	0	1.0000	1.0000	NaN	NaN
T13	0	1.0000	1.0000	NaN	NaN
ENV1	0	0.0000	0.0000	NaN	NaN
ENV2	2	0.0000	0.0000	1.00	2.00
ENV3	6	0.0000	0.0000	1.00	5.00
ENV4	0	-3.0000	5.0000	NaN	NaN
ENV5	0	-3.5000	8.5000	NaN	NaN
ENV6	5	2.5000	6.5000	1.00	2.00
ENV7	0	-4.1250	6.8750	NaN	NaN
ENV8	0	0.0000	8.0000	NaN	NaN

Table 2.6 Z-Score Outlier Detection Results (threshold = 3)

Column	Outliers Count	Min Outlier	Max Outlier
Project_No	0	NaN	NaN
Simple Actors	2	4.00	4.00
Average Actors	0	NaN	NaN
Complex Actors	0	NaN	NaN
UAW	0	NaN	NaN
Simple UC	1	20.00	20.00
Average UC	0	NaN	NaN
Complex UC	0	NaN	NaN
UUCW	0	NaN	NaN
TCF	0	NaN	NaN
ECF	2	0.51	0.51
Real_P20	0	NaN	NaN
Real_Effort_Person_Hours	0	NaN	NaN
T1	4	4.00	5.00
T2	0	NaN	NaN
T3	0	NaN	NaN
T4	0	NaN	NaN
T5	0	NaN	NaN
T6	0	NaN	NaN
T7	0	NaN	NaN
T8	0	NaN	NaN
T9	0	NaN	NaN
T10	5	0.00	0.00
T11	2	3.00	3.00
T12	0	NaN	NaN
T13	0	NaN	NaN
ENV1	0	NaN	NaN
ENV2	2	1.00	2.00
ENV3	1	5.00	5.00
ENV4	0	NaN	NaN
ENV5	0	NaN	NaN
ENV6	2	1.00	1.00
ENV7	0	NaN	NaN
ENV8	2	0.00	0.00

While some values occasionally lie outside the expected range as defined by the IQR and Z-score thresholds, its magnitude is not sufficiently extreme to warrant removal. More importantly, given the limited number of instances in the dataset (only 70 projects), removing even a single row could disproportionately impact model training and validation, especially under cross-validation regimes. Therefore, it was decided to retain the outlier in the dataset.

This conservative approach prioritizes data retention over aggressive cleaning, especially in the context of small datasets where each sample carries significant statistical weight. Furthermore, the presence of this outlier is not expected to unduly bias the models, as later steps such as standardization and regularization help mitigate the influence of such values. Thus, the dataset is deemed suitable for modeling without the need for outlier exclusion.

Conclusions from Exploratory Data Analysis (EDA) Based on the exploratory data analysis conducted in the previous sections, several conclusions can be drawn regarding the suitability of the dataset for predictive modeling in the context of software effort estimation:

- **Feature richness:** The dataset contains a sufficiently diverse set of features that are conceptually and empirically relevant to software effort estimation. These include both raw indicators (e.g., number of actors, technical/environmental factors) and compound metrics (e.g., UAW, UUCW, TCF, ECF) derived from established estimation methodologies such as Use Case Points (UCP). This diversity ensures that the models will have access to both granular and aggregated views of the problem domain.
- **Data completeness:** The final selection of features used for modeling contains no missing values. Although the raw dataset included missing entries in some categorical columns, these columns were either excluded during feature selection or had no impact on the best-performing feature subsets. As a result, imputation or row removal was not required, preserving the integrity and size of the dataset.
- **Normalization requirements:** Due to the heterogeneous scale and distribution of feature values—ranging from small integer counts to continuous scores—normalization is necessary to ensure fair model training. Standardization using `StandardScaler` was

selected as the preferred scaling method. Importantly, the same normalization procedure is applied consistently across all models, including both linear regressors and deep neural networks, in order to prevent any methodological bias that could arise from preprocessing discrepancies.

- **Outlier tolerance:** The total number of detected outliers is minimal, and their deviation from the mean or interquartile range is not sufficiently severe to justify removal. Retaining these outliers ensures that no additional information is lost—an important consideration given the limited sample size. Moreover, the downstream modeling pipeline includes techniques (e.g., regularization and robust loss functions) that help mitigate the influence of outliers, further supporting the decision to retain them.

Taken together, the EDA confirms that the dataset is well-structured, informative, and appropriately preprocessed for subsequent application of both traditional regression models and deep learning techniques. The insights obtained from this phase also inform key modeling decisions, such as feature scaling, choice of evaluation metrics, and the rationale for not employing data imputation or trimming.

2.4 Feature Description

In order to construct reliable predictive models for software effort estimation, it is crucial to understand the meaning, role, and real-world relevance of the features present in the dataset. This subsection provides a detailed description of all features, connecting them with established practices in software engineering, particularly the Use Case Points (UCP) method.

Actors and Use Cases The dataset includes counts of actors and use cases classified by complexity:

- **Simple Actors, Average Actors, Complex Actors:** In the UCP method, actors are external entities interacting with the system. They are categorized based on the complexity of the interaction:
 - *Simple Actor:* Interacts via an API (Weight = 1)

- *Average Actor*: Interacts via a protocol like TCP/IP (Weight = 2)
- *Complex Actor*: Interacts through a graphical user interface (GUI) (Weight = 3)
- **Simple UC, Average UC, Complex UC**: Use cases describe functional requirements. Complexity is based on the number of steps:
 - *Simple Use Case*: 1–3 steps (Weight = 5)
 - *Average Use Case*: 4–7 steps (Weight = 10)
 - *Complex Use Case*: 8+ steps (Weight = 15)

[6]

Aggregated Metrics

- **UAW** (Unadjusted Actor Weight): Computed as the sum of each type of actor multiplied by its corresponding weight.
- **UUCW** (Unadjusted Use Case Weight): Computed similarly for use cases.

Together, **UAW** and **UUCW** form the foundation for calculating the unadjusted size of the project, which reflects the system's functional size before technical and environmental adjustments.

Adjustment Factors Adjustment factors adjust the unadjusted size according to the project's technical and environmental realities:

- **TCF** (Technical Complexity Factor): Accounts for technical requirements and challenges (e.g., distributed systems, performance, reusability). Calculated based on weighted evaluations of 13 technical factors.
- **ECF** (Environmental Complexity Factor): Reflects the working environment, team skills, and stability. Calculated based on 8 environmental factors.

Both factors are multiplicative and adjust the unadjusted use case points (UUCP) to generate a more realistic size estimate.

Target Variables

- **Real Effort (Person-Hours):** The true effort required for project completion. This is the main target variable for predictive modeling.

Categorical Project Metadata

- **Sector, Language, Methodology, ApplicationType, DataDonator:** These fields provide categorical information about the context of each project. They may affect effort indirectly by correlating with system complexity, size, or team experience.

Technical Complexity Factors (T1–T13) The Technical Complexity Factors (TCFs) reflect the technical requirements and characteristics that influence project complexity and effort. Each factor is rated typically on a scale from 0 (irrelevant) to 5 (essential). The factors are:

- **T1: Distributed system** — Degree to which the system is distributed across multiple locations, networks, or platforms.
- **T2: Performance objectives** — Importance of meeting specific performance requirements, such as response time or throughput.
- **T3: End-user efficiency** — Extent to which the system must optimize the user's efficiency and productivity.
- **T4: Complex internal processing** — Degree of complexity in the internal processing logic or calculations within the system.
- **T5: Reusability** — Requirement for developing reusable components that may be used in other systems or modules.

- **T6: Easy to install** — Need for the system to be simple and straightforward to install and configure.
- **T7: Easy to use** — Emphasis on intuitive user interfaces and system usability to minimize training and support.
- **T8: Portability** — Need for the system to be portable across different hardware platforms or operating systems.
- **T9: Ease of change** — Importance of designing the system to facilitate maintenance and future enhancements.
- **T10: Concurrent use** — Degree to which the system must support multiple users accessing and modifying shared data simultaneously.
- **T11: Security features** — Requirement for implementing access controls, data protection, and other security measures.
- **T12: Access for third parties** — Need for external systems, APIs, or third-party components to access or integrate with the system.
- **T13: Special user training** — Extent to which special or advanced user training is required for effective use of the system.

Each factor is evaluated typically on a 0–5 scale, and weighted according to its predefined importance.

Environmental Complexity Factors (ENV1–ENV8) Each ENV1–ENV8 field corresponds to environmental aspects such as:

- **ENV1: Familiarity with the development process** — Extent to which the team is experienced with the used development processes and standards.
- **ENV2: Application experience** — Degree of the team's experience with the type of application being developed (e.g., finance, healthcare).

- **ENV3: Object-oriented experience** — Level of the team's familiarity and experience with object-oriented analysis and design principles.
- **ENV4: Lead analyst capability** — Skill level and effectiveness of the lead analyst coordinating requirements and analysis activities.
- **ENV5: Motivation** — Overall motivation and commitment of the project team members.
- **ENV6: Stable requirements** — Degree to which the project's requirements are well-defined and stable throughout the development lifecycle.
- **ENV7: Part-time staff** — Proportion of staff working part-time on the project; high part-time involvement can lower efficiency.
- **ENV8: Difficult programming language** — Difficulty and unfamiliarity associated with the programming language(s) used in the project.

Each environmental factor influences the overall productivity and efficiency of the development team.

Relation to Real-World Estimation Practices This dataset is structured according to the Use Case Points (UCP) methodology, a common technique in software estimation that translates functional size (use cases and actors) into expected effort through adjustment factors (technical and environmental). This approach has been widely used in industry due to its simplicity, relative accuracy, and reliance on early-stage artifacts (use cases).

In a real-world scenario:

- **Actor and Use Case complexity** reflect how intricate the interactions and functionality will be.
- **Technical factors** adjust for the technical landscape challenges.
- **Environmental factors** adjust for human/team and organizational factors.

Together, they provide a reasonable early estimate of software development effort.

Use of Weighting Factors in Modeling When we use the compound features we use the UCP method weights as they are used to calculate the compound metric. However when we use the individual features we let the model determine the weights on its own. This can be beneficial as it was determined that in some cases Neural networks can outperform the classical weighted formula used by UCP method [20].

When selecting features we can try both methods, as well as using the compound metrics as inputs, as well as using the individual features. We cant however use both as an input as colinearity is detrimental to both regression and neural networks[11].

Grid search using several feature selection options was used to determine which feature selection offers best performance.

2.5 Feature Selection

Feature selection plays a critical role in predictive modeling, as it directly affects both the accuracy and generalizability of the resulting models. In the present study, several considerations were made when determining which variables to include in the regression models.

To begin with, two variables were explicitly excluded from all feature selection candidates. The first, DataDonator, represents the source of each project record. Including this variable could introduce spurious correlations or overfitting to source-specific artifacts, as it does not encode any intrinsic project property relevant to effort estimation. The second variable, Real_P20, was excluded because it is a derived target-like variable that is highly correlated (perfectly in percentile terms) with the primary response variable Real_Effort_Person_Hours. Including such a variable would constitute data leakage and artificially inflate model performance.

As a starting point (baseline), we used the set of compound metrics commonly employed in the Use Case Points (UCP) methodology—namely, UAW (Unadjusted Actor Weight), UUCW (Unadjusted Use Case Weight), TCF (Technical Complexity Factor), and ECF (Environmental

Complexity Factor). This configuration reflects industry practice and is often adopted when simple effort estimation is required early in the project lifecycle.

To assess whether predictive performance could be improved by introducing more granular feature representations, a systematic grid search was performed. In this search, compound features were progressively broken down into their constituent components. For instance, in the *Actor breakdown* configuration, the compound variable UAW was replaced with its underlying components—Simple Actors, Average Actors, and Complex Actors—while excluding UAW to avoid multicollinearity. The same approach was applied to use cases, technical factors (T1–T13), and environmental factors (ENV1–ENV8).

The results of the grid search, using linear regression as the base model, are illustrated in Figure 2.4.

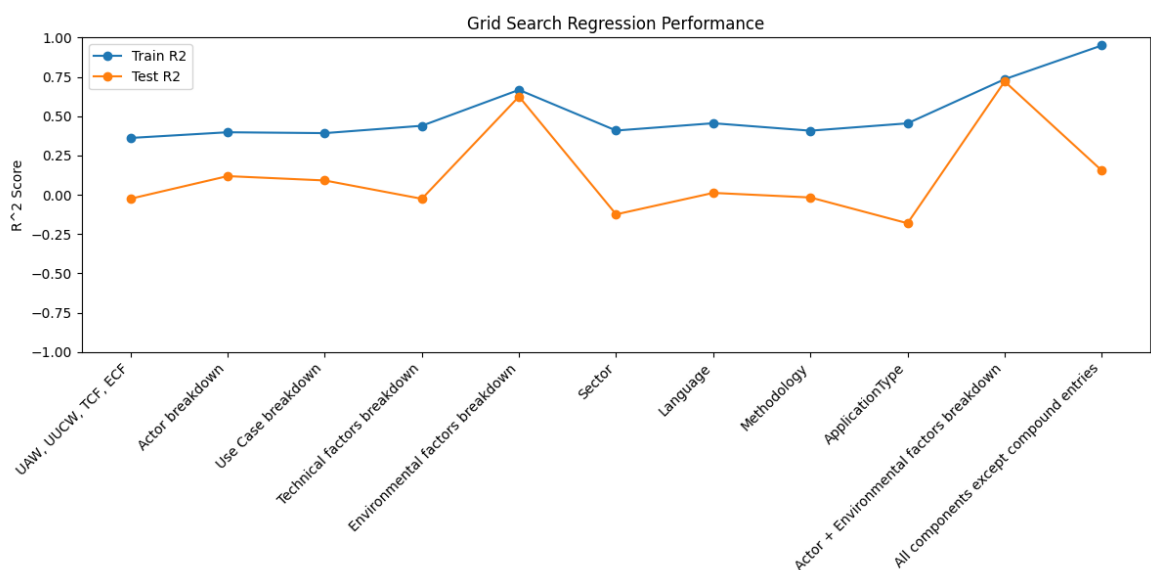


Figure 2.4 Grid search results for feature selection using linear regression. Configurations include baseline UCP metrics and various decompositions of compound features into their raw components.

Best performance was found for feature Selection: Simple Actors, Average Actors, Complex Actors, UUCW, TCF, ENV1, ENV2, ENV3, ENV4, ENV5, ENV6, ENV7 a ENV8.

Each configuration along the x-axis represents a distinct feature set, where certain compound metrics are replaced by their decomposed forms. The y-axis shows the R^2 score for both training and test sets, enabling an evaluation of performance and potential overfitting.

The results indicate that decomposing the environmental complexity factors had the most significant positive impact on model performance. This suggests that the individual environmental indicators (e.g., team experience, familiarity with process, stability of requirements) carry predictive information that may be diluted when aggregated into a single index. The actor breakdown also contributed a modest performance gain, and importantly, the combination of both breakdowns yielded the highest R^2 on the test set—implying an additive effect of feature granularity.

Interestingly, expanding the use case or technical factors alone did not lead to marked improvements, and in some configurations, they slightly degraded performance, possibly due to increased dimensionality without sufficient variance or signal. To prevent overfitting and multicollinearity, each time a compound feature was decomposed, it was removed from the feature set to avoid redundant encoding of the same underlying information.

These findings underscore the importance of thoughtful feature engineering, particularly in domains where compound metrics are prevalent. They also support the hypothesis that finer-grained representations of environmental and actor-related factors are more informative than their aggregated counterparts, at least in the context of linear modeling.

2.6 Data Preprocessing

Before the dataset can be utilized for training predictive models, a comprehensive data preprocessing pipeline must be applied. Proper preprocessing ensures that the models receive clean, consistent, and appropriately structured data, which is particularly critical in the context of small datasets such as the one used in this study. The preprocessing steps executed in this work are summarized below:

- **Feature selection:** As discussed in Section 2.5, an extensive grid search was conducted to identify the most informative set of input variables. The best-performing feature combinations were selected based on regression performance, ensuring relevance of the feature set.
- **Handling missing values:** The final selected features contain no missing values, thus

eliminating the need for imputation. This is a favorable outcome, as it avoids introducing artificial values into the dataset and preserves all available data points. Given the limited size of the dataset, this is particularly beneficial, since discarding even a few samples could negatively impact the model's ability to generalize.

- **Data type validation:** All columns in the dataset were verified to have appropriate data types. No inconsistencies (e.g., numerical features stored as strings or mixed types) were found, and therefore no type conversions were necessary.
- **One-hot encoding of categorical variables:** Although one-hot encoding was performed during the feature selection experiments for completeness, the categorical variables involved were ultimately excluded from the best-performing feature subsets.
- **Feature scaling:** Due to the presence of numerical features with widely varying scales, normalization was required. Standardization was performed using `StandardScaler`, which transforms the features to have zero mean and unit variance. This step is essential for both regression models and neural networks, as unscaled features can lead to poor convergence.
- **Collinearity control:** To prevent multicollinearity, particular care was taken to avoid including both compound metrics and their respective component variables simultaneously. For example, if individual actor categories (Simple, Average, Complex) were included, the aggregate variable UAW was excluded. This ensures that models are not fed redundant or highly correlated information, which could lead to inflated coefficients or unstable training.
- **Outlier handling:** As previously discussed, a dedicated outlier analysis showed that while a few mild outliers exist, they do not exert sufficient influence to warrant removal. The overall number and magnitude of these outliers are small, and keeping them helps maintain the size and variance of the dataset.
- **Train/Test split:** Given the small number of samples, splitting the dataset into training and testing subsets must be done with caution to avoid compromising statistical power. These considerations, along with the specific validation strategy employed (e.g., k -fold cross-validation), are elaborated on in later chapters.

It is important to highlight that both regression models and neural networks share similar requirements with respect to data preprocessing. Consequently, a single unified preprocessing pipeline was used for all modeling experiments. This decision was motivated not only by technical compatibility but also by the desire to ensure a fair comparison between the two modeling approaches. Applying identical preprocessing minimizes the risk of introducing preprocessing-induced bias that could artificially benefit one model family over the other. This contributes to the methodological soundness of the comparative analysis.

2.6.1 Data Preprocessing implementation

Before training the machine learning models, the dataset underwent a series of preprocessing steps to ensure data quality and improve model performance. This section describes the complete preprocessing pipeline, including handling of missing values, encoding of categorical variables, and scaling of numerical features.

Overview of the Pipeline. The preprocessing steps were implemented using `scikit-learn`'s `Pipeline` and `ColumnTransformer` utilities, which allow for reproducible and modular transformation of different feature types. The original data was loaded from a CSV file, irrelevant columns were removed, and features were separated into numerical and categorical subsets:

```
1 df = pd.read_csv('data_orig.csv')
2 columns_to_drop = ['Real_P20', 'Project_No', 'DataDonator']
3 df = df.drop(columns=columns_to_drop)
4
5 X = df.drop(['Real_Effort_Person_Hours'], axis=1)
6 y = df['Real_Effort_Person_Hours']
```

Listing 2.1 Loading and splitting the raw dataset

Handling Missing Values. Two different strategies were used for imputing missing values: (1) for **numerical features**, missing values were filled using the *median* value of each column; and (2) for **categorical features**, the *most frequent* value (mode) was used. This was implemented using `SimpleImputer` within dedicated transformation pipelines:

```
1 numeric_transformer = Pipeline(steps=[
2     ('imputer', SimpleImputer(strategy='median')),
3     ('scaler', StandardScaler()))
4 ])
5
6 categorical_transformer = Pipeline(steps=[
7     ('imputer', SimpleImputer(strategy='most_frequent')),
8     ('onehot', OneHotEncoder(handle_unknown='ignore',
9     sparse_output=False)))
9 ])
```

Listing 2.2 Numeric and categorical imputers

Scaling of Numerical Features. Scaling was applied to ensure that all numeric features contribute equally to the model. Specifically, the StandardScaler was used, which transforms each feature to have zero mean and unit variance:

$$x_{\text{scaled}} = \frac{x - \mu}{\sigma}$$

where μ is the mean and σ is the standard deviation of the feature.

Standardization is essential for neural networks and gradient-based optimizers because it:

- Prevents features with larger magnitudes from dominating the learning process
- Accelerates convergence during training
- Reduces numerical instability

Encoding Categorical Variables. Categorical variables were transformed using **One-Hot Encoding**, which replaces each categorical variable with a set of binary indicator columns. For example, a column DevelopmentType with values Agile, Waterfall, and Hybrid becomes three binary columns:

[DevType_Agile, DevType_Waterfall, DevType_Hybrid]

Unknown categories during inference are safely ignored using handle_unknown='ignore' to prevent runtime errors.

Final Data Export. The transformed numerical and one-hot encoded categorical features were combined into a single matrix using ColumnTransformer, and saved as a new CSV file:

```

1 preprocessor = ColumnTransformer(transformers=[
2     ('num', numeric_transformer, numeric_features),
3     ('cat', categorical_transformer, categorical_features)
4 ])
5
6 X_processed = preprocessor.fit_transform(X)
7
8 categorical_features_processed = preprocessor.
9     named_transformers_['cat'].named_steps['onehot'].
10    get_feature_names_out(categorical_features)
11 feature_names = np.concatenate([numeric_features,
12    categorical_features_processed])
13
14 X_processed_df = pd.DataFrame(X_processed, columns=
15     feature_names)
16 X_processed_df['Real_Effort_Person_Hours'] = y.values
17 X_processed_df.to_csv('data_preprocessed.csv', index=False)

```

Listing 2.3 Applying preprocessing and saving the result

Summary of Preprocessing Steps.

- Imputed missing values (median for numeric, mode for categorical)
- Scaled numeric features to standard normal distribution
- One-hot encoded all categorical variables
- Combined all transformed features into a clean matrix

These preprocessing steps ensured the dataset was in an optimal format for machine learning and neural network modeling, improving the stability and interpretability of results.

Scaling visualisation Table 2.7 illustrates the transformation of raw feature values after applying standard scaling. The original values for the actor counts are in their natural units, while the scaled values have been transformed to have zero mean and unit variance. This normalization ensures that all input features contribute comparably during model training, regardless of their original scale. As shown, the absolute magnitudes have shifted and are

Table 2.7 Three example entries for actor counts before and after scaling.

	Simple Actors	Average Actors	Complex Actors
Before scaling 1. row	1.000000	1.000000	2.000000
Before scaling 2. row	1.000000	2.000000	1.000000
Before scaling 3. row	0.000000	0.000000	3.000000
After scaling 1. row	0.318626	0.065198	-0.420596
After scaling 2. row	0.318626	1.206166	-1.089725
After scaling 3. row	-0.796566	-1.075769	0.248534

now centered around zero, which is a desired property for neural network optimization algorithms.

2.7 Evaluation Metrics and Validation Strategy

Accurate and unbiased evaluation of model performance is essential for drawing valid conclusions in predictive modeling, particularly in comparative studies involving multiple model types. In the context of software effort estimation, where datasets tend to be small and heterogeneous, choosing appropriate metrics and validation strategies becomes even more critical. This section outlines the evaluation metrics used in this work, along with the cross-validation approach adopted to ensure statistical reliability of the results.

2.7.1 Performance Metrics

To assess model performance from both explanatory and error-based perspectives, two widely used regression metrics were employed: the coefficient of determination (R^2) and the Root Mean Squared Error (RMSE). These metrics provide complementary insights into the predictive capabilities of the models.

Coefficient of Determination (R^2) The coefficient of determination, denoted as R^2 , is a widely used metric in regression analysis that quantifies how well the model explains the variance of the dependent (target) variable. [11] It is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

where:

- y_i represents the actual observed values,
- $\hat{y}_i$ denotes the model's predicted values,
- $\bar{y}$ is the mean of the observed values,
- n is the number of samples in the dataset.

The numerator of the formula represents the residual sum of squares (RSS), i.e., the unexplained variance, while the denominator represents the total sum of squares (TSS), i.e., the variance in the data. The resulting ratio therefore expresses the proportion of variance that remains unexplained by the model relative to the total variance.

An R^2 value of 1.0 indicates that the model predictions perfectly match the actual outcomes, meaning 100% of the variance is explained. A score of 0.0 implies that the model performs no better than using the mean of the target variable as a predictor. Negative values are also possible and occur when the model's predictions are worse than this naive mean-based baseline.

In the context of software effort estimation, a higher R^2 indicates that the model effectively captures the variability in project effort across different types of software projects. However, it should be noted that R^2 is sensitive to outliers and does not convey information about the scale of prediction errors, which is why it is typically used alongside other metrics such as RMSE.

Root Mean Squared Error (RMSE) The Root Mean Squared Error (RMSE) is a frequently used metric that quantifies the average magnitude of prediction errors in a regression model. It is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Here, y_i denotes the actual values, $\hat{y}_i$ the predicted values, and n the total number of observations. The RMSE represents the square root of the mean of the squared differences between actual and predicted values. [12]

Because the errors are squared before averaging, RMSE gives a relatively high weight to large errors. This makes it particularly sensitive to outliers and useful in scenarios where large deviations from the true values are especially undesirable. The result is expressed in the same units as the target variable, making it directly interpretable in practical terms.

In the domain of software effort estimation, where the target variable is typically measured in person-hours, a lower RMSE implies that the model produces predictions that are closer, on average, to the actual recorded efforts. This metric is particularly helpful in determining whether a model's estimates are within an acceptable range for real-world project planning and resource allocation.

2.7.2 Validation Methods

Train-Test Split One of the most fundamental approaches for evaluating model performance is the train-test split, where the dataset is partitioned into two disjoint subsets:

- **Training set:** A subset of the data used to fit or train the predictive model.
- **Testing set:** A held-out subset used solely to evaluate the generalization ability of the trained model.

While this method is simple, fast, and computationally efficient, it suffers from significant limitations when applied to small datasets. Specifically, partitioning an already limited number of samples reduces the amount of data available for training, which may lead to underfitting and high variance in model performance. The evaluation metrics obtained from a single split can be heavily dependent on which particular samples happen to be in the training or testing set, resulting in an unreliable estimate of true model generalization. [21]

Despite these limitations, train-test splitting remains useful in the early stages of exploratory analysis, where fast feedback is needed to experiment with different modeling strategies. In

such cases, preliminary results obtained from a basic split can help guide decisions about model architecture, feature engineering, or hyperparameter ranges. These initial insights can then be validated more rigorously using cross-validation methods.

K-Fold Cross-Validation To obtain more reliable and robust estimates of model performance—especially in small datasets—this study employs K-Fold Cross-Validation (K-Fold CV), a resampling technique that offers a more comprehensive use of the data. The process proceeds as follows:

- The entire dataset is divided into K equally sized subsets, known as "folds."
- In each of the K iterations, one fold is held out as the test set, while the model is trained on the remaining $K - 1$ folds.
- The model is evaluated on the held-out fold, and the performance metric (e.g., R^2 , RMSE) is recorded.
- After all K iterations, the individual scores are averaged to produce the final performance estimate.

K-Fold CV provides multiple advantages over the simple train-test split. First, it ensures that every sample in the dataset is used for both training and testing, thereby maximizing data efficiency. Second, it mitigates the variance in performance metrics caused by arbitrary data partitioning, resulting in more stable and generalizable evaluation results. Third, the method provides insight into the variability of model performance across different subsets of data, which can be useful for assessing robustness. [12]

In this study, K-Fold CV is used consistently across all modeling experiments to ensure methodological fairness and comparability. A 3-fold cross-validation scheme was selected as a compromise between computational efficiency and evaluation reliability, given the relatively small number of data points (70 projects). The mean and standard deviation of the performance metrics across folds are reported to capture both central tendency and dispersion of the model's predictive ability. mates of model performance compared to a single train-test split.

2.8 Implementation

This section describes the practical implementation of regression models and deep neural networks (DNNs) used for the prediction of software development effort. Particular attention is paid to the tools, libraries, and design choices made to ensure a robust and reproducible modeling process.

2.8.1 Software and Libraries Used

For the development and evaluation of all models, the Python programming language was selected due to its extensive ecosystem for machine learning and data science.

The primary library utilized was **Scikit-learn** (sklearn), a widely adopted open-source machine learning library that offers:

- **A comprehensive suite of models:** Scikit-learn provides ready-to-use implementations of a wide variety of machine learning algorithms, including linear models, regularized regressions, support vector machines, decision trees, and ensemble methods.
- **Unified and consistent API:** All models share a simple, consistent interface with `fit()`, `predict()`, and `score()` methods, facilitating quick experimentation and comparability.
- **Integrated preprocessing tools:** Scikit-learn includes preprocessing capabilities such as scaling, normalization, encoding, missing value imputation, and feature selection.
- **Pipeline and model selection frameworks:** Scikit-learn provides `Pipeline` and `GridSearchCV` classes to automate complex workflows, model training, and hyperparameter tuning in a reproducible manner.
- **Cross-validation utilities:** Extensive tools for cross-validation, including `train-test split` and `standard k-fold` which are essential for our validation strategies.

In the context of this thesis, Scikit-learn was particularly advantageous because:

- Many regression algorithms could be instantiated with a few lines of code, using carefully optimized and reliable implementations.
- Hyperparameter search and model selection were greatly simplified using `GridSearchCV` in combination with cross-validation, reducing the potential for errors in manual validation procedures.

For the implementation of deep neural networks, additional specialized libraries were used alongside Scikit-learn.

Specifically, the **TensorFlow** framework and its high-level API **Keras** were utilized. TensorFlow is an open-source machine learning platform developed by Google, offering extensive support for deep learning tasks, including model building, optimization, and deployment. Keras, which is tightly integrated with TensorFlow, provides a user-friendly, high-level interface for defining, training, and evaluating deep neural networks with minimal boilerplate code.

The combination of TensorFlow and Keras offers several advantages:

- **Flexible model architecture definition:** Neural networks can be defined either sequentially or using more complex graph-based approaches, allowing full control over depth, width, activation functions, and connectivity.
- **Advanced optimization algorithms:** TensorFlow includes state-of-the-art optimizers (e.g., Adam, RMSprop) essential for efficient neural network training.
- **Support for GPU acceleration:** TensorFlow can leverage GPU hardware to significantly speed up training times for large and deep models.
- **Integrated callbacks and monitoring:** Features such as early stopping, learning rate scheduling, and checkpointing facilitate efficient and safe model training.

While Scikit-learn provided a strong foundation for traditional regression models and simple machine learning pipelines, TensorFlow and Keras were essential to address the greater complexity and flexibility required by neural networks.

The overall choice of Python, Scikit-learn, TensorFlow, and Keras enabled rapid development, extensive experimentation, and rigorous evaluation of both traditional and deep learning models while adhering to the best practices of modern scientific computing and machine learning research.

2.8.2 Regression Models Implementation

Dataset Handling and Target Specification The dataset used in this study was loaded from a structured file named `data.csv`, located in the same directory as the modeling script.

Grid Search Over Feature Selections and Regression Models As previously discussed in Section 2.5, feature selection was conducted using a systematic grid search over various combinations of raw and compound variables derived from the UCP methodology. This step yielded several promising candidate feature sets based on their performance with linear regression.

To further assess the robustness and generalizability of these results, we extended the grid search by applying additional regression models, including polynomial regression, Lasso regression, and Ridge regression. Each model introduces distinct assumptions and regularization mechanisms:

- **Polynomial regression** captures non-linear relationships by introducing higher-order terms.
- **Lasso regression** (L1-regularized) performs both feature selection and shrinkage, which is beneficial when dealing with multicollinearity or redundant variables.
- **Ridge regression** (L2-regularized) penalizes large coefficient magnitudes, improving generalization in the presence of noisy or highly correlated features.

Grid Search Implementation

Feature Set Selection. As discussed in previous chapters, the feature selection strategy was designed to align with established practices in software effort estimation, while also taking into account the need to avoid multicollinearity.

These feature sets were implemented in code as follows:

```

1 feature_sets = [
2     ("UAW, UUCW, TCF, ECF", ['UAW', 'UUCW', 'TCF', 'ECF']),
3     ("Actor breakdown", ['Simple Actors', 'Average Actors', '
4         Complex Actors', 'UUCW', 'TCF', 'ECF']),
5     ("Use Case breakdown", ['UAW', 'Simple UC', 'Average UC',
6         'Complex UC', 'TCF', 'ECF']),
7     ("Technical factors breakdown", ['UAW', 'UUCW', 'ECF'] +
8         [f'T{i}' for i in range(1, 14)]),
9     ("Environmental factors breakdown", ['UAW', 'UUCW', 'TCF'
10         ] + [f'ENV{i}' for i in range(1, 9)]),
11     ("Sector", ['UAW', 'UUCW', 'TCF', 'ECF'] + [col for col
12         in df.columns if col.startswith('Sector_')]),
13     ("Language", ['UAW', 'UUCW', 'TCF', 'ECF'] + [col for col
14         in df.columns if col.startswith('Language_')]),
15     ("Methodology", ['UAW', 'UUCW', 'TCF', 'ECF'] + [col for
16         col in df.columns if col.startswith('Methodology_')]),
17     ("ApplicationType", ['UAW', 'UUCW', 'TCF', 'ECF'] + [col
18         for col in df.columns if col.startswith('
19         ApplicationType_')]),
20     ("Actor + Environmental factors breakdown",
21         ['Simple Actors', 'Average Actors', 'Complex Actors', '
22             UUCW', 'TCF'] +
23         [f'ENV{i}' for i in range(1, 9)]),
24     ("All components except compound entries",
25         ['Simple Actors', 'Average Actors', 'Complex Actors',
26         'Simple UC', 'Average UC', 'Complex UC'] +
27         [f'T{i}' for i in range(1, 14)] +
28         [f'ENV{i}' for i in range(1, 9)] +
29         [col for col in df.columns if col.startswith(('Sector_',
30             'Language_', 'Methodology_', 'ApplicationType_'))
31         and not col.startswith('DataDonator_')])

```

Listing 2.4 Definition of feature sets for grid search

Model Configuration. To evaluate the performance of classical regression techniques, four different models were configured and tested using grid search to optimize hyperparameters:

- **Linear Regression:** A baseline model with no hyperparameters to tune. It was included to serve as a reference for comparison with more complex models.
- **Polynomial Regression:** Implemented as a pipeline combining `PolynomialFeatures` with `LinearRegression`. Degrees 2 and 3 were tested via grid search (`poly__degree`

= [2, 3]) to capture potential non-linear relationships without excessive model complexity. Higher degrees were avoided due to the risk of overfitting, especially given the small dataset size.

- **Lasso Regression:** A linear model with L1 regularization, evaluated with different values of the regularization parameter α = [0.001, 0.01, 0.1, 1.0]. These values were selected to span a range from minimal to relatively strong regularization, allowing assessment of how much sparsity improves performance.
- **Ridge Regression:** Similar to Lasso but using L2 regularization, tested with the same α values. This provided insight into whether L1 or L2 regularization better fits the structure of the data.

All models were integrated into a parameter grid and optimized using GridSearchCV. The search was conducted over the training data, and the best parameter set for each model was selected based on cross-validated performance.

```

1 models = {
2     'Linear Regression': {
3         'model': LinearRegression(),
4         'params': {}
5     },
6     'Polynomial Regression': {
7         'model': Pipeline([
8             ('poly', PolynomialFeatures()),
9             ('regressor', LinearRegression())
10        ]),
11        'params': {
12            'poly__degree': [2, 3]
13        }
14    },
15    'Lasso Regression': {
16        'model': Lasso(),
17        'params': {
18            'alpha': [0.001, 0.01, 0.1, 1.0]
19        }
20    },
21    'Ridge Regression': {
22        'model': Ridge(),
23        'params': {
24            'alpha': [0.001, 0.01, 0.1, 1.0]
25        }
26    }
27 }

```

Listing 2.5 Full model and parameter grid configuration used in GridSearchCV

This setup enabled a systematic and fair comparison across models, capturing linear, poly-

nomial, and regularized variants, while maintaining computational feasibility.

Validation Strategy. For each feature set, the dataset was first split into a training set (80%) and a test set (20%) using a hold-out approach.

After the best model was identified, it was evaluated on the previously held-out test set using two key regression metrics: the coefficient of determination (R^2) and the Root Mean Squared Error (RMSE). The following snippet shows the relevant evaluation loop:

```

1 grid_search = GridSearchCV(model_info['model'], model_info['
  params'],
2                               cv=5, scoring='
                                neg_mean_squared_error', n_jobs
                                =-1)
3 grid_search.fit(X_train, y_train)
4 best_model = grid_search.best_estimator_
5 y_pred = best_model.predict(X_test)
6 rmse = np.sqrt(mean_squared_error(y_test, y_pred))
7 r2 = r2_score(y_test, y_pred)

```

Listing 2.6 Evaluation loop with inner CV and test set scoring

Results Aggregation and Visualization. The average and standard deviation of R^2 and RMSE were recorded for each (model, feature set) pair. These were stored in separate dataframes and visualized using heatmaps, as shown in Figure 2.5 and Figure 2.6

Summary. This experimental setup enabled a thorough investigation of how feature engineering interacts with model architecture. The use of nested grid search inside a cross-validation loop ensured robust selection of the best-performing configuration. The results show that certain feature subsets, especially those incorporating environmental and categorical context, can significantly improve predictive performance when paired with appropriate regularization or non-linear models.

All models were evaluated using two primary performance metrics: the coefficient of determination (R^2) and Root Mean Squared Error (RMSE), as discussed in earlier sections. This comparative modeling experiment not only aids in identifying the best-performing model

configuration, but also strengthens confidence in the selected feature set before benchmarking against more complex architectures such as deep neural networks.

The complete results of this regression grid search are visualized using heatmaps, which offer an intuitive representation of performance across models and feature combinations. These figures serve as the foundation for selecting the final candidate to be evaluated against neural network models in subsequent sections.



Figure 2.5 Feature Selection and model grid search using R^2



Figure 2.6 Feature Selection and model grid search using RMSE

Model and Feature Selection Results Based on the results obtained from the regression grid search, the most effective feature configurations were those involving the breakdown of compound variables—specifically, the environmental factors and the combination of actor and environmental factors. These configurations consistently outperformed the baseline (which uses aggregated compound metrics such as UAW and ECF), suggesting that valuable predictive information is lost when using aggregated forms of these variables. Disaggregating the compound features appears to provide the model with a more granular understanding of project complexity and team conditions, leading to improved prediction accuracy.

Regarding the performance of the regression models, notable differences emerged depending on the selected feature set. Linear regression, Lasso regression, and Ridge regression demonstrated strong performance only when applied to the best-performing feature sets. However, on most other feature selections, their performance dropped significantly, often approaching $R^2 \approx 0$, indicating poor generalization. In contrast, polynomial regression did not achieve top performance on the best feature sets but exhibited more consistent behavior across a wider range of feature configurations. This suggests that while it may not capture the strongest signals in optimal settings, it generalizes more steadily across varied inputs.

Although Ridge regression performed poorly on many feature configurations—often yielding results close to $R^2 = 0$ —it achieved the highest accuracy on the two strongest feature sets. In contrast, linear and Lasso regression models similarly underperformed on most configurations and only produced competitive results under optimal conditions. Polynomial regression did not outperform others on the top-performing sets but demonstrated more consistent behavior across different feature subsets.

Ridge regression was ultimately selected as the final model for further analysis due to its peak performance on the most informative feature selections. Although its variability across configurations raises interesting questions about feature interaction and model robustness, these fall outside the current scope and may warrant future study. For the purposes of this work, we consider the results sufficient to justify continuing with Ridge regression as the most effective model under the best conditions.

The selected feature set for final evaluation includes:

- UUCW (Use Case Weight),
- TCF (Technical Complexity Factor),
- A breakdown of Actors into Simple, Average, and Complex categories,
- A breakdown of Environmental Factors into their constituent components.

This configuration was selected based on its slightly superior performance across multiple metrics and feature configurations when compared to other models, while maintaining robustness and interpretability.

K-Fold Cross-Validation To ensure the robustness and generalizability of the final model, an additional evaluation was conducted using K -fold cross-validation. This step serves to reduce the impact of random partitioning and to ensure a fair comparison with the neural network model, which will be evaluated under the same conditions. This technique partitions the data into k subsets (folds), trains the model on $k - 1$ folds, and tests it on the remaining one. This is repeated k times, each time changing the test fold.

To evaluate model generalization performance and prevent overfitting, **K-Fold Cross-Validation** was used. This technique partitions the data into k subsets (folds), trains the model on $k - 1$ folds, and tests it on the remaining one. This is repeated k times, each time changing the test fold.

```

1 from sklearn.model_selection import KFold
2 from sklearn.metrics import mean_squared_error, r2_score
3 import numpy as np
4
5 kf = KFold(n_splits=3, shuffle=True, random_state=42)
6
7 rmse_scores = []
8 r2_scores = []
9
10 for train_index, test_index in kf.split(X):
11     X_train, X_test = X[train_index], X[test_index]
12     y_train, y_test = y[train_index], y[test_index]
13
14     model.fit(X_train, y_train)
15     y_pred = model.predict(X_test)
16
17     rmse = mean_squared_error(y_test, y_pred, squared=False)
18     r2 = r2_score(y_test, y_pred)
19
20     rmse_scores.append(rmse)
21     r2_scores.append(r2)
22
23 print("Mean RMSE:", np.mean(rmse_scores))
24 print("Mean R2:", np.mean(r2_scores))

```

Listing 2.7 3-Fold Cross-Validation Implementation in Python

Explanation

The above code performs 3-fold cross-validation using the `KFold` class from `scikit-learn`. Key steps are as follows:

- `KFold(n_splits=3)` divides the dataset into three equally sized folds.
- `shuffle=True` ensures the data is randomly shuffled before splitting, to avoid order bias.
- `model.fit()` trains the regression model on the training folds.
- `model.predict()` is used to make predictions on the current test fold.
- `mean_squared_error(..., squared=False)` computes the Root Mean Squared Error (RMSE).
- `r2_score(...)` computes the coefficient of determination R^2 .
- The process is repeated three times and the mean RMSE and R^2 values are reported to assess overall model performance.

This evaluation strategy provides a compromise between robustness and computational efficiency, and is suitable for smaller data

In this process, the selected Ridge regression model was trained and evaluated using 3-fold cross-validation, with the chosen feature subset as input. The results are reported in Table 2.8, summarizing the Root Mean Squared Error (RMSE) and R^2 score for each fold, as well as the overall mean and standard deviation for both metrics. This provides a stable estimate of model performance and helps validate the consistency of the modeling pipeline.

Table 2.8 K-Fold Cross-Validation Results for Ridge Regression

Fold	RMSE	R ² Score
Fold 1	363.883	0.700
Fold 2	575.913	0.216
Fold 3	377.655	0.670
Overall (Mean ± Std)	439.150 ± 96.869	0.529 ± 0.221

As shown in Table 2.8, the average performance of the Ridge regression model across the three folds yields an R^2 score of 0.529 and an RMSE of approximately 439.15 person-hours. While these figures represent a noticeable decrease in performance compared to earlier hold-out validation results, this outcome is not unexpected. Given the small size of the dataset, individual train/test splits can introduce considerable variability, and performance is often inflated when models are evaluated on a single favorable split.

Despite the drop in average performance, the adoption of K-Fold cross-validation provides a more robust and statistically reliable estimate of model generalization. It ensures that each data point contributes to both training and testing phases and mitigates the bias introduced by a single partition. Moreover, the standard deviation values (± 96.87 for RMSE and ± 0.221 for R^2) offer insight into the stability of the model across different subsets of the data.

Consequently, this cross-validated result will serve as the official benchmark for the regression model and will be used as the baseline for evaluating the neural network model, which will undergo the same K-Fold validation procedure to ensure a fair and consistent comparison.

2.8.3 Deep Neural Networks Implementation

In implementing a deep neural network (DNN) model for software effort estimation, we aimed to follow best practices reported in recent literature while adapting the architecture to suit the constraints of our dataset. The design choices were guided by practical experimentation tailored to the nature and size of our data.

The mainly used architecture in effort estimation is a feed-forward neural network (FFNN). Regarding network depth, several studies have demonstrated that a single hidden layer is

often sufficient to capture the relevant patterns in structured data, even for relatively complex tasks. Although deeper architectures—using two or three hidden layers—have shown some improvements in specific cases, these benefits are typically observed in the presence of large datasets, which help mitigate the risk of overfitting[22].

The width of each hidden layer, defined as the number of neurons per layer, was selected based on experimental heuristics. Prior research suggests that increasing the number of neurons beyond a moderate threshold (e.g., 10–15 nodes) yields diminishing returns in terms of prediction accuracy, especially in low-data regimes. In line with this, our experiments confirmed that relatively narrow layers performed better on our limited dataset [23].

Historically, activation functions such as sigmoid and hyperbolic tangent ($\tanh$) were prevalent in hidden layers. However, in modern practice the Rectified Linear Unit (ReLU) becomes more common due to its computational efficiency and favorable gradient properties. Accordingly, we employed ReLU activation in all hidden layers [24]. For the output layer—tasked with producing a continuous effort estimate—a single neuron with a linear activation function was used, as is standard for regression tasks [25].

Initial attempts using deeper and wider networks resulted in poor performance and unstable training. This observation highlighted a common issue in small datasets: high-capacity models tend to overfit and fail to generalize. Once the architecture was simplified by reducing both the number of layers and the total number of neurons, the models produced viable predictions. This reinforces findings in the literature that even relatively shallow architectures can be effective when matched to the scale and complexity of the data [26].

Grid Search for Neural Network Configuration To determine the optimal configuration for the neural network, a grid search was performed across key architectural and training hyperparameters. The objective was to identify a model that offers strong predictive performance while remaining compact and computationally efficient—critical considerations given the small size of the dataset (70 samples) and the need to avoid overfitting.

The hyperparameters explored included:

- Number of neurons in the hidden layer: {4, 8},
- Learning rate: {0.01, 0.1},
- L2 regularization factor: {0.01, 0.1},
- Dropout rate: {0.2, 0.4},
- Batch normalization: {True, False},
- Batch size: 16 (fixed),
- Activation function: ReLU (fixed).

This setup resulted in 32 unique hyperparameter combinations, each evaluated using 3-fold cross-validation—ensuring consistency with the evaluation protocol used for regression models. Training was limited to a maximum of 50 epochs, with early stopping enabled (patience = 10) to reduce overfitting and avoid unnecessary computation. However, in practice, early stopping was rarely triggered, and most training runs completed the full 50 epochs.

The grid could have been extended in several dimensions, but its design reflects a deliberate trade-off between computational feasibility and model relevance, given the small dataset and the desire to limit the risk of overfitting. The number of hidden units was restricted to {4, 8} neurons. Architectures with fewer neurons (e.g., 2) were excluded based on the assumption that they would lack sufficient capacity to model the complexity of the data, while larger configurations (e.g., 16) were avoided due to overfitting concerns and increased parameter counts, which would not be justified given only 70 training samples.

The learning rate grid included only two values: {0.01, 0.1}. These are higher than the commonly recommended default for the Adam optimizer (0.001 [27]), but were selected based on initial trials where fast convergence was observed, and because they allowed the grid search to focus on interactions between regularization, dropout, and batch normalization. We acknowledge that including lower learning rates such as 0.001 or 0.005 could have offered smoother training dynamics and possibly improved generalization, and should be considered in future work.

For L2 regularization, only two values were tested: {0.01, 0.1}. This limited range was intended to cover weak to moderate regularization strength while avoiding very small values (e.g., 0.001) that might be negligible at the given network scale. A finer resolution (e.g., including 0.005) would likely be beneficial in future studies with more computational bandwidth.

Dropout was tested at two levels: 0.2 and 0.4. These values were chosen to represent typical mild-to-moderate regularization settings for small fully connected layers. While narrower values such as 0.1 or 0.3 may have provided a more refined view of regularization effects, they were omitted to keep the grid size tractable. This simplification is acknowledged as a limitation.

The batch size was fixed at 16 across all experiments. This choice balances stable gradient estimation with training speed, which is particularly relevant for small datasets. Although exploring smaller (e.g., 8) or larger (e.g., 32) batches might reveal differences in convergence or generalization, this dimension was fixed to reduce experimental complexity.

Batch normalization, where used, was applied **after** the activation function (ReLU). The effect of normalization on weight distributions and internal covariate shift was not explicitly measured, but this would be a valuable direction for future interpretability studies.

Overall, the grid search was intentionally constrained to focus on interactions between a few critical architectural and training hyperparameters, while keeping the total number of configurations within computationally manageable bounds. This design enabled meaningful comparisons while acknowledging that broader search spaces could yield improved results.

The complete grid is summarized in Table 2.9, and results were compared using both R^2 and RMSE. The configuration yielding the best average performance across folds was selected for final comparison against the best-performing regression model.

Each configuration is passed to the model creation function, which dynamically builds a neural network using the specified parameters.

```
1 def create_model(input_dim, config):  
2     model = Sequential()
```

Table 2.9 Neural Network Grid Search Parameters

Parameter	Values
Number of Neurons	{4, 8}
Learning Rate	{0.01, 0.1}
L2 Regularization	{0.01, 0.1}
Dropout Rate	{0.2, 0.4}
Batch Size	{16}
Activation Function	{ReLU}
Batch Normalization	{True, False}
Total Configurations	32
Cross-Validation Folds	3
Maximum Epochs	50
Early Stopping Patience	10

```

3     model.add(Dense(config['neurons'], activation=config['
4         activation'], input_dim=input_dim, kernel_regularizer=
5             l2(config['l2_reg'])))
6     if config['use_batch_norm']:
7         model.add(BatchNormalization())
8     model.add(Dropout(config['dropout']))
9     model.add(Dense(1)) # Output layer
10    model.compile(optimizer=Adam(learning_rate=config['
11        learning_rate']),
        loss='mse', metrics=['mae'])
    return model

```

Listing 2.8 Model creation with dynamic configuration

Parallel Execution and Result Sorting To speed up the evaluation process, each configuration is processed in parallel using Python's multiprocessing module. The best configurations are ranked by their average R^2 scores.

```

1 with mp.Pool(processes=mp.cpu_count()) as pool:
2     process_config_partial = partial(process_config, X=X, y=y
3         , y_scaler=y_scaler)
4     results = list(tqdm(pool.imap(process_config_partial,
5         configs), total=len(configs)))
6 results.sort(key=lambda x: x['mean_r2'], reverse=True)

```

Listing 2.9 Parallel evaluation and result sorting

The grid search approach enabled an exhaustive evaluation of various neural network ar-

chitectures, balancing complexity, regularization, and normalization. The final model was selected based on the highest average R^2 score across validation folds.

From the hyperparameter grid search, the best-performing neural network configuration was selected for final evaluation. Its performance was assessed using 3-fold cross-validation to maintain consistency with the regression evaluation procedure. The results are presented in Table 2.10.

Table 2.10 K-Fold Cross-Validation Results for Neural Network

Fold	RMSE	R ² Score
Fold 1	264.83	0.8411
Fold 2	352.15	0.7069
Fold 3	306.58	0.7825
Overall (Mean ± Std)	307.85 ± 35.66	0.7769 ± 0.0549

As seen in Table 2.10, the neural network exhibits strong and stable performance across all three folds. The average R^2 score of 0.7769 significantly exceeds that of the best regression model (0.529), indicating a superior ability to capture variance in the target variable. Moreover, the RMSE is consistently lower across all folds, further reinforcing the model’s predictive advantage.

Notably, this high performance was achieved with a relatively simple architecture. In contrast, larger networks—those with more neurons or deeper structures—frequently failed to generalize and, in several cases, yielded predictions worse than the dataset mean. This underscores the importance of architectural simplicity and regularization when working with limited data.

The configuration of the best-performing neural network is summarized in Table 2.11.

This configuration balances model complexity with regularization and is well-suited to the small size of the dataset. It serves as the final benchmark against which all other models in this study are compared.

The neural network configuration that achieved the best performance during the evaluation process is illustrated in Figure 2.7 below.

Table 2.11 The configuration of the best-performing neural network

Parameter	Value
Input Layer	13 features
Hidden Layer 1	4 neurons with ReLU activation
Dropout Rate	0.2
Output Layer	1 neuron (linear activation)
Optimizer	Adam
Learning Rate	0.1
Loss Function	Mean Squared Error (MSE)
Regularization	L2 (0.01)
Batch Size	16
Epochs	50 (with early stopping)
Early Stopping Patience	10
Cross-Validation	3-fold
Random Seed	42

Figure 2.7 provides a structural visualization of the neural network architecture that achieved the best performance during hyperparameter tuning. This architecture is relatively simple, consisting of a single hidden layer with 4 neurons, ReLU activation, and L2 regularization. A dropout layer with a rate of 0.2 is applied after the hidden layer to reduce the risk of overfitting. The final output layer contains a single neuron with a linear activation function, which is appropriate for continuous regression tasks.

Despite its simplicity, this neural network outperformed more complex architectures in terms of R^2 score and prediction stability. This suggests that, for the given dataset and feature set, a lightweight model with proper regularization and training strategy is sufficient to capture the underlying relationships without overfitting.

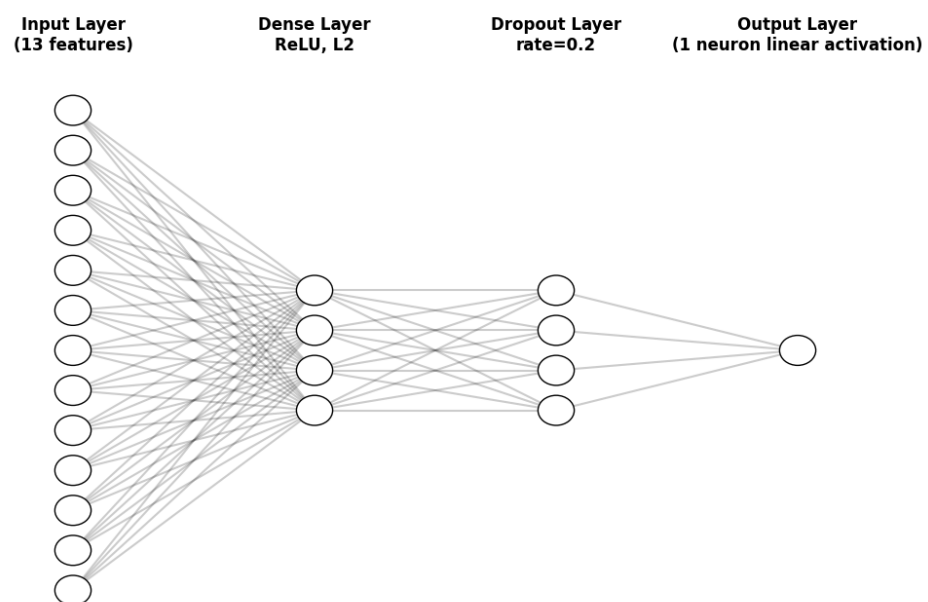


Figure 2.7 Architecture of best performing neural network

3 RESULTS

The comparative results between the best-performing regression model (Ridge Regression) and the best-performing neural network are summarized in Table 3.1. Performance was evaluated using 3-fold cross-validation to ensure a robust and fair comparison, given the small dataset size.

Table 3.1 Comparison of K-Fold Cross-Validation Results Between Ridge Regression and Neural Network

Fold	RMSE (Regression)	RMSE (NN)	R^2 (Regression)	R^2 (NN)
Fold 1	363.88	264.83	0.70	0.84
Fold 2	575.91	352.15	0.22	0.71
Fold 3	377.66	306.58	0.67	0.78
(Mean $\pm$ Std)	439.15 $\pm$ 96.87	307.85 $\pm$ 35.66	0.53 $\pm$ 0.22	0.78 $\pm$ 0.05

As shown in Table 3.1, the neural network outperformed Ridge Regression across all three validation folds. On average, the neural network achieved an RMSE of 307.85 compared to 439.15 for the regression model, representing a reduction in prediction error of approximately 30%. Likewise, the neural network attained a significantly higher mean R^2 score (0.7769) relative to the regression model (0.529), suggesting that the neural network was better able to explain the variance in the target variable.

It is worth noting that not only did the neural network demonstrate superior average performance, but it also exhibited lower standard deviation across the folds for both metrics. This suggests more stable and consistent predictions, even when trained and tested on different subsets of the data—an especially important trait in low-sample settings.

The substantial performance gap between the models becomes even more notable when considering the simplicity of the final neural network architecture. Despite employing only a single hidden layer with four neurons, the network was able to model nonlinear interactions in the data that traditional regression methods could not capture. This outcome aligns with findings in prior research, where neural networks have been shown to outperform linear and even regularized regression methods in the domain of software effort estimation.

Interestingly, the regression model's performance degraded significantly in Fold 2, with an

R^2 of just 0.216 and an RMSE exceeding 570. This fold-specific instability further reinforces the limitations of traditional regression under limited data conditions. In contrast, the neural network maintained more consistent results across all folds, indicating better generalization capabilities.

These results not only confirm the effectiveness of neural networks for this type of task but also emphasize the importance of hyperparameter tuning and architectural simplification in small-data environments. They also suggest that compound features—when broken into their elemental components—can better inform nonlinear models than linear ones, due to the neural network’s capacity to extract subtle and complex interactions.

3.1 Visual Comparison of Performance

Error Behavior and Diagnostic Visualization To complement the numerical evaluation metrics such as R^2 and RMSE, we generated a set of diagnostic plots for both the neural network and Ridge regression models. These include boxplots and histograms of prediction errors, as well as scatter plots comparing predicted vs. actual values and residuals versus predicted values.

Error Distribution and Outliers. From the boxplots of prediction errors, we observe that the neural network exhibits a more compact interquartile range (IQR) and a tighter central mass of residuals (Figure 3.1). The median error for the neural network is slightly negative, suggesting a mild tendency to underestimate effort.

In contrast, Ridge regression is more symmetrically distributed around zero but with a significantly higher spread and more severe outliers (Figure 3.2). One notable outlier in Ridge regression exceeds +2000 person-hours, which is not present in the neural network’s distribution. This implies that while Ridge may have lower average bias, it is more susceptible to extreme mispredictions. The neural network demonstrates more stability and consistency across most data points.

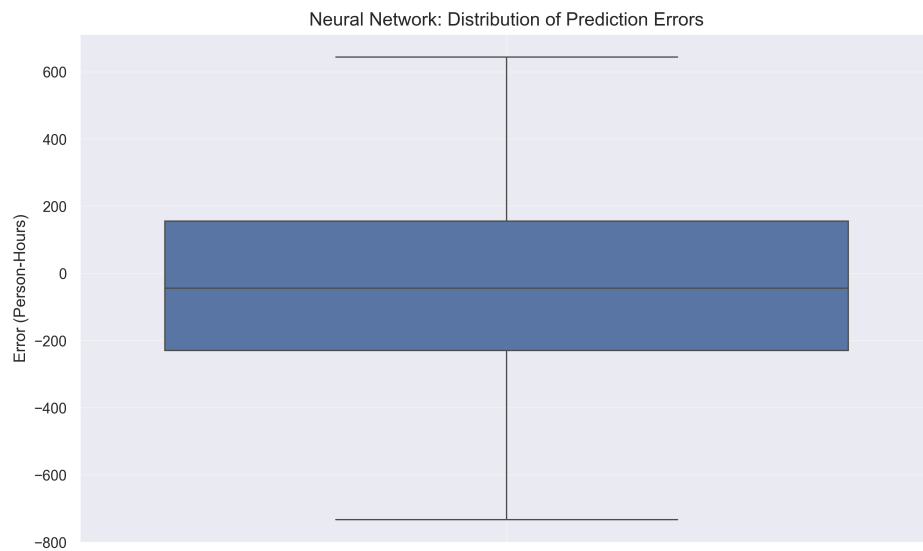


Figure 3.1 Neural network: Distribution of prediction errors.

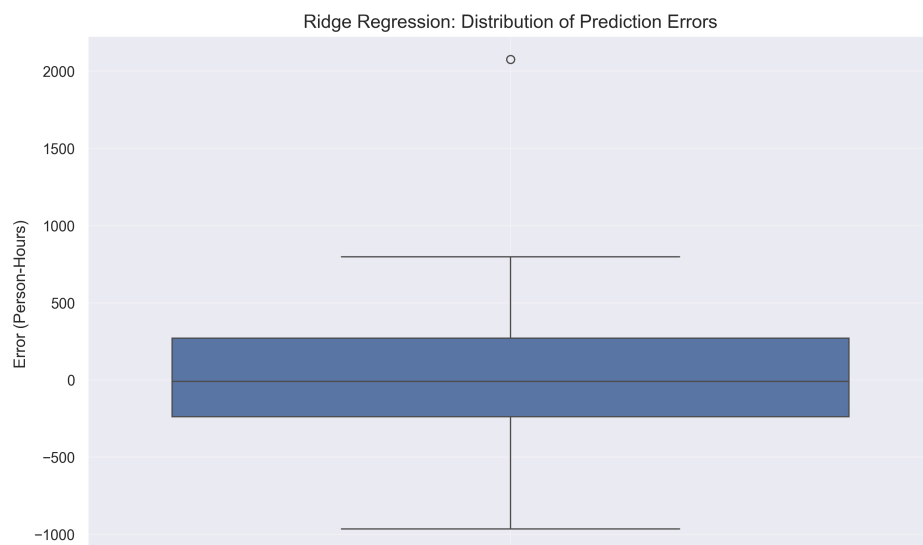


Figure 3.2 Ridge regression: Distribution of prediction errors.

Error Distribution Shape. The histograms with kernel density estimation provide insight into the overall shape of error distributions. The neural network’s error distribution is slightly left-skewed, confirming its tendency to underestimate (Figure 3.3). Ridge regression shows a more Gaussian-like profile but with heavier tails and a wider spread, indicating higher volatility in predictions (Figure 3.4).

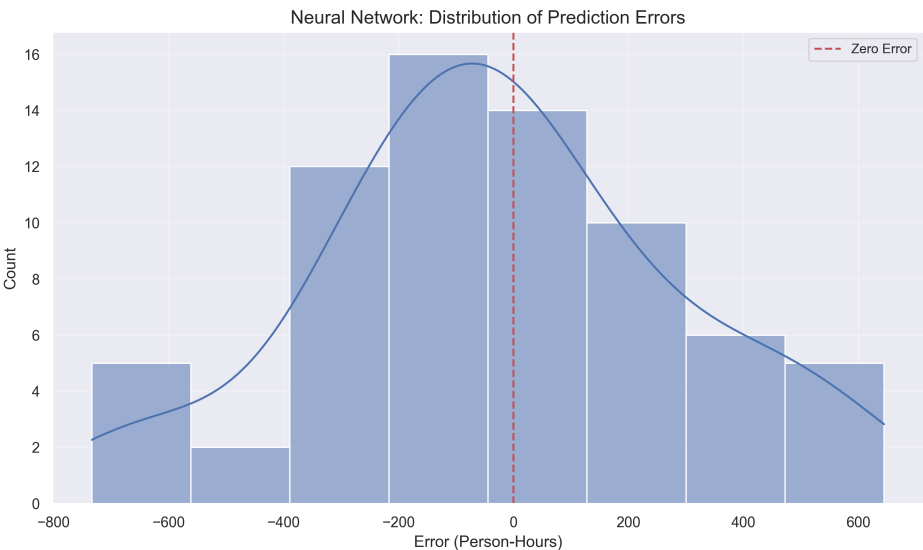


Figure 3.3 Neural network: Histogram of prediction errors with KDE.

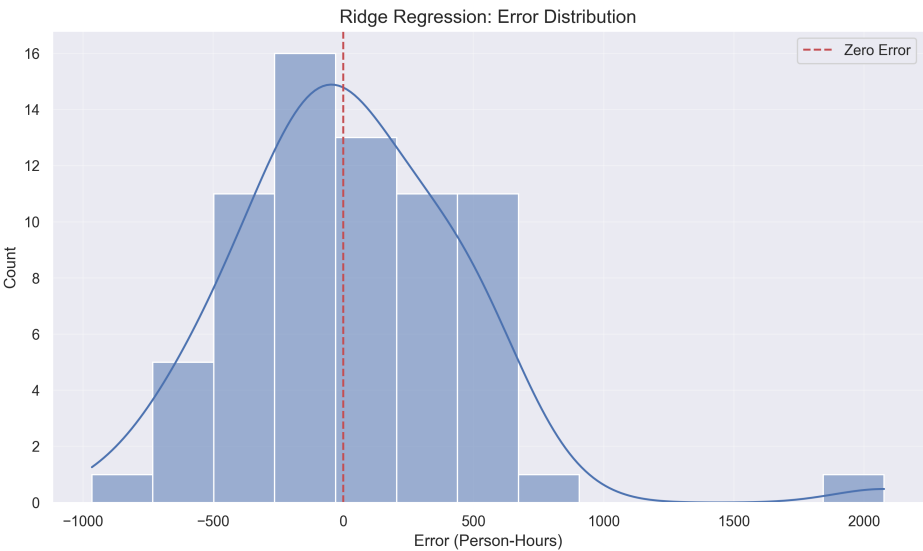


Figure 3.4 Ridge regression: Histogram of prediction errors with KDE.

Error vs Predicted Value. The scatter plots of prediction error vs. predicted values reveal systematic patterns. The Ridge model shows a clear funnel shape (Figure 3.6), suggesting heteroscedasticity—errors grow as predicted values increase. This violates key regression assumptions and may indicate poorer performance on large projects.

By contrast, the neural network displays a more homogeneous error spread (Figure 3.5), which supports its robustness across varying project sizes.

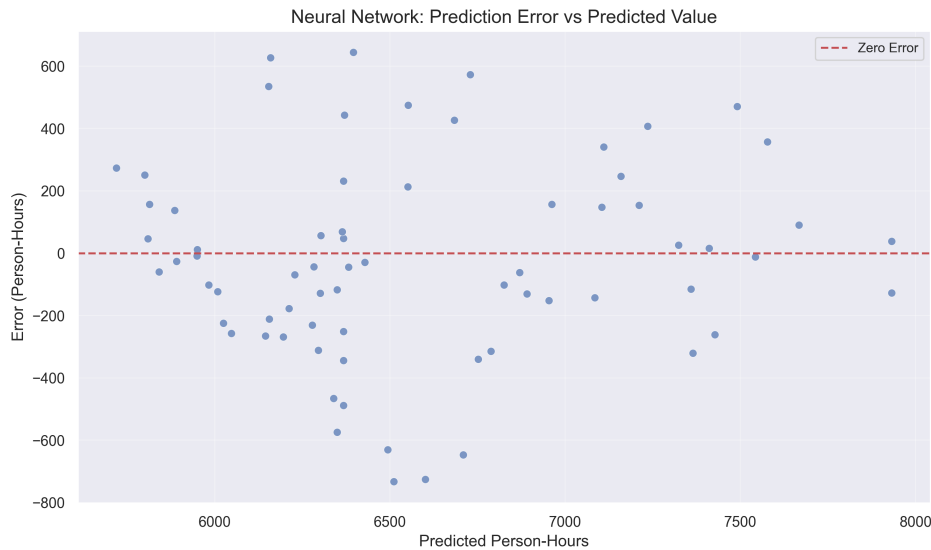


Figure 3.5 Neural network: Error vs predicted value.

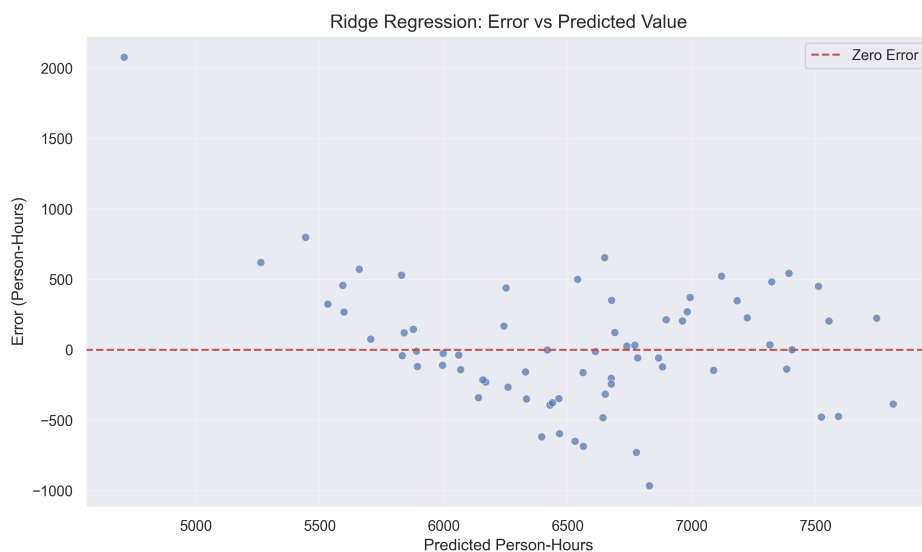


Figure 3.6 Ridge regression: Error vs predicted value.

Predicted vs Actual Alignment. Finally, the predicted vs actual plots emphasize model alignment with real values. The neural network (Figure 3.7) shows adherence to the diagonal, indicating reliable approximation. Ridge regression (Figure 3.8), while showing some alignment, demonstrates a looser pattern with systematic deviations, especially overestimating low values and underestimating high ones.

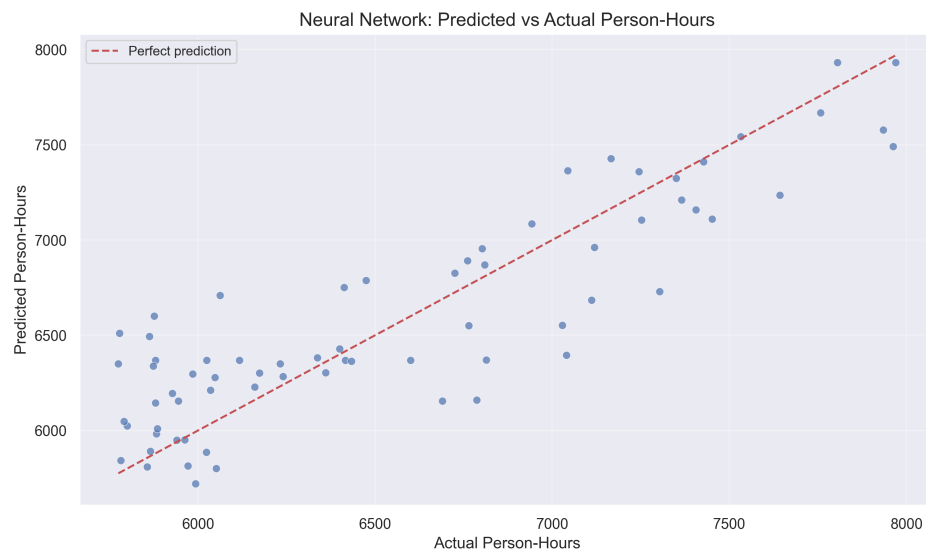


Figure 3.7 Neural network: Predicted vs actual person-hours.

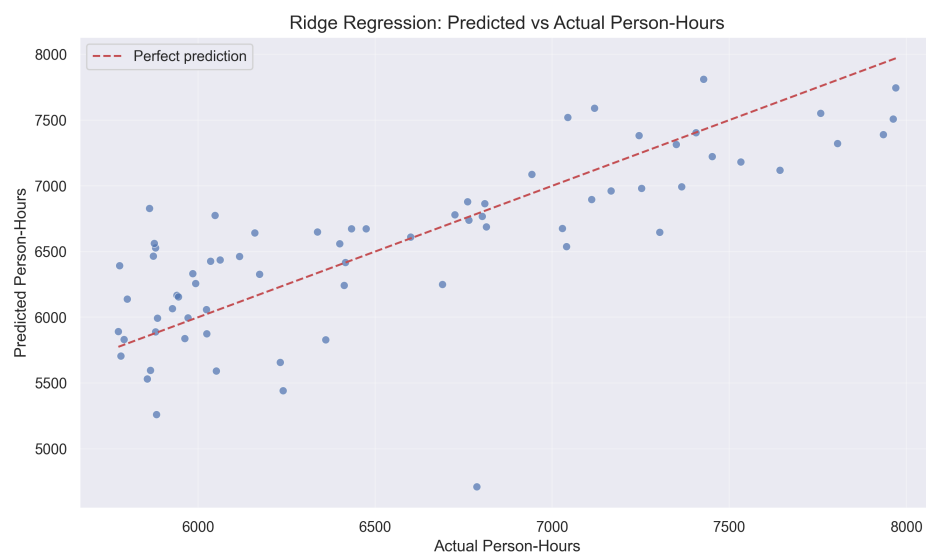


Figure 3.8 Ridge regression: Predicted vs actual person-hours.

Conclusion. Both models offer reasonable baseline performance, but the visualizations show that the neural network delivers more consistent results, with less error variation and better generalization across projects of different sizes. In contrast, Ridge regression—while simpler and easier to interpret—exhibits more variability, occasional outliers, and signs of heteroscedasticity. These limitations suggest that it may be less reliable in diverse software estimation scenarios where project characteristics vary significantly.

4 CONCLUSION AND FUTURE WORK

This thesis compared several standard regression approaches—Linear, Ridge, Lasso, and Polynomial regression—with a feedforward deep neural network, using a real-world dataset for software effort estimation. All models were trained under the same preprocessing pipeline and evaluated via 3-fold cross-validation, with feature selection and tuning done through grid search.

The key result of this work is that even a small neural network—with a single hidden layer and just four neurons—clearly outperformed all tested regression models. It achieved a reduction in RMSE of over 130 person-hours and improved the R^2 score by approximately 25 percentage points compared to the best-performing regression baseline. This confirms that even compact neural architectures, when correctly tuned, can capture complex, non-linear relationships in structured estimation data that traditional models miss.

From a practical standpoint, the simplicity of the winning neural model is particularly valuable. It required minimal computational resources and was fast to train, yet delivered significantly better predictions. This is especially relevant for teams working with limited data or tooling. However, the experiment also suggests that scaling to larger models and datasets could unlock even greater performance. The current results likely represent only a lower bound on what neural networks could achieve in this domain.

Limitations. The dataset used consisted of only 70 samples, which naturally limited the scope of architectures explored. To avoid overfitting, the chosen neural network was deliberately small. A broader evaluation—both in terms of model size and training strategies—was constrained by the dataset size. Additionally, the grid search was kept narrow to manage training time, meaning better hyperparameter combinations may remain undiscovered. No external dataset was used for validation, so the generalizability of results remains to be tested.

Future Work. Several directions could build on these findings:

- **Larger datasets and deeper models:** With more data, deeper networks could be explored without overfitting, potentially capturing subtler patterns and improving gen-

eralization.

- **More exhaustive hyperparameter search:** Expanding the grid search to include a wider range of learning rates, dropout rates, and architectural variants may reveal better-performing configurations.
- **Interpretability via structured neural design:** One promising direction is to map neurons or layers to interpretable concepts from established effort estimation models (e.g., actors, use cases, or technical complexity factors). This could make neural predictions more transparent and actionable.
- **Model ensembles and hybrid systems:** Combining neural networks with decision trees, boosting methods, or linear models may improve robustness and offer confidence intervals on predictions.
- **External validation and transfer learning:** Testing the model on different datasets—or pretraining on related tasks before fine-tuning—could reveal how adaptable these models are across projects or domains.

In conclusion, this work confirms that even on small datasets, neural networks can outperform traditional regression models in the task of software effort estimation—provided that their architecture is carefully constrained to avoid overfitting. While existing literature already demonstrates the superiority of deep learning models over regression techniques in large-scale scenarios, our findings extend this advantage to smaller, UCP-based datasets as well. By conducting a thorough grid search, we identified specific neural network configurations—such as a single hidden layer with four neurons—that achieved consistently better performance than any tested regression alternative. These results suggest that with appropriate tuning, neural networks represent a practical and effective tool for improving estimation accuracy, even in data-limited real-world software development environments.

REFERENCES

- [1] MCCONNELL, Steve, 2006. *Software Estimation: Demystifying the Black Art*. Microsoft Press. ISBN 0-7356-0535-1.
- [2] TRENDOWICZ, Adam and JEFFERY, Ross, 2014. *Software Project Effort Estimation: Foundations and Best Practice Guidelines for Success*. Springer. ISBN 978-3-319-03629-8. Available from: [10.1007/978-3-319-03629-8](https://link.springer.com/book/10.1007/978-3-319-03629-8). Available from: <https://link.springer.com/book/10.1007/978-3-319-03629-8>.
- [3] BOEHM, Barry and ABTS, Chris, 2000. *Software Cost Estimation with COCOMO II*. Prentice Hall. ISBN 978-0137025763.
- [4] KELLEHER, John D.; NAMEE, Brian Mac and D'ARCY, Aoife, 2015. *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. The MIT Press. ISBN 978-0-262-02944-5.
- [5] DAMODARAN, Mohan, 2002. A Case Study of Software Effort Estimation using the Use Case Points Method. In: *Proceedings of the 19th Annual Conference for Information Systems Educators (ISECON)*. Č. 253d. Available from: <https://iscap.us/proceedings/isecon/2002/253d/ISECON.2002.Damodaran.pdf>.
- [6] BOER, A., 2007. Use Case Points. https://www.researchgate.net/publication/220018983_Use_Case_Points. Accessed: 10-May-2025.
- [7] REQTEST, 2023. Agile Estimation Techniques: A True Estimation Process. Available from: <https://reqtest.com/en/knowledgebase/agile-estimation-techniques/>. Accessed: 2025-06-01.
- [8] GOURAV KARMAKAR, Varun Gupta, 2019. Agile Story Points Estimation using Machine Learning. *Journal of Software Engineering Research and Development*. 2019. 7, num. 1, p. 1–12.
- [9] BOEHM, Barry; ABTS, Chris; WINSOR BROWN, A.; CHULANI, Sunita; CLARK, Bert et al., 2000. *COCOMO II Model Definition Manual*. Center for Software Engineering, University of Southern California. Available from: https://athena.ecs.csus.edu/~buckley/CSc231_files/Cocomo_II_Manual.pdf. Accessed: 2025-06-01.
- [10] GARMUS, David and HERRON, David, 2001. *Function Point Analysis: Measurement Practices for Successful Software Projects*. Addison-Wesley. ISBN 978-0201699440. Available from: <https://www.amazon.com/Function-Point-Analysis-Measurement-Successful/dp/0201699443>.
- [11] JAMES, Gareth; WITTEN, Daniela; HASTIE, Trevor and TIBSHIRANI, Robert, 2021. *An Introduction to Statistical Learning: With Applications in R*. Springer Texts in Statistics. Springer. Second edition. ISBN 978-1-0716-1417-4.
- [12] HASTIE, Trevor; TIBSHIRANI, Robert and FRIEDMAN, J. H., 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer. Second edition. ISBN 978-0-387-84857-0.
- [13] GOODFELLOW, Ian; BENGIO, Yoshua and COURVILLE, Aaron, 2016. *Deep Learning*. Adaptive Computation and Machine Learning Series. The MIT Press. ISBN 978-0-262-03561-3.
- [14] NASSIF, Ali Bou; AZZEH, Mohammad; CAPRETZ, Luiz Fernando and HO, Danny, 2016. Neural Network Models for Software Development Effort Estimation: A Comparative Study. *arXiv preprint arXiv:1611.09934*. 2016. Available from: <https://arxiv.org/abs/1611.09934>.
- [15] MURPHY, Kevin P., 2012. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. MIT Press. ISBN 978-0-262-01802-9.

- [16] AYGUN, Abdullah; DAL, Hasan and DALKILIÇ, Yunus Emre, 2024. A Systematic Review of Software Development Effort Estimation Studies Using Machine Learning. In: *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. p. 237–246. SCITEPRESS. Available from: 10.5220/0013072300003693. Available from: <https://www.scitepress.org/Papers/2024/130723/130723.pdf>.
- [17] BREIMAN, Leo, 2001. Random Forests. *Machine Learning*. 2001. 45, num. 1, p. 5–32. Available from: 10.1023/A:1010933404324. Available from: <https://doi.org/10.1023/A:1010933404324>.
- [18] ALJAHDALI, Sultan; ALSAQER, Mohammad; ALABDULWAHAB, Areej S.; ALSUBAIE, Khlood F.; ALWAGAIT, Nawal S. et al., 2015. Estimating Software Effort and Function Point Using Regression, Support Vector Machine and Artificial Neural Networks Models. In: *2015 12th International Conference on Information Technology - New Generations*. p. 507–512. Available from: 10.1109/ITNG.2015.90. Available from: <https://ieeexplore.ieee.org/document/7507149/>.
- [19] KUMAR, S. and MALHOTRA, R., 2022. Comparative Analysis of Machine Learning Techniques in Effort Estimation. In: *2022 12th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. p. 639–644. Available from: 10.1109/Confluence52989.2022.9850592. Available from: <https://ieeexplore.ieee.org/document/9850592/>.
- [20] RANKOVIĆ, Nevena; RANKOVIĆ, Dragica; IVANOVIĆ, Mirjana and LAZIĆ, Ljubomir, 2021. A novel UCP model based on artificial neural networks and orthogonal arrays. *Applied Sciences*. 2021. 11, num. 19, p. 8799. Available from: 10.3390/app11198799. Available from: <https://doi.org/10.3390/app11198799>.
- [21] MÜLLER, Andreas C. and GUIDO, Sarah, 2016. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media. ISBN 9781449369415.
- [22] JUN, Seong and LEE, Jae, 2023. Deep Learning Models for Software Effort Estimation. *Systems*. 2023. 11, num. 2, p. 91. Available from: <https://www.mdpi.com/2079-8954/11/2/91>.
- [23] ELGENDY, Mohamed; TOLBA, Safa and ZAYED, Mohamed, 2024. Effect of the Number of Hidden Layer Neurons on the Accuracy of the Back Propagation Neural Network. https://www.researchgate.net/publication/377755645_Effect_of_the_Number_of_Hidden_Layer_Neurons_on_the_Accuracy_of_the_Back_Propagation_Neural_Network. Accessed: 2025-06-01.
- [24] AGARAP, Abien Fred, 2018. Deep Learning using Rectified Linear Units (ReLU). *arXiv preprint*. 2018. Available from: <https://arxiv.org/abs/1803.08375>.
- [25] ENTHOUGHT, 2021. Understanding the Output Layer of a Neural Network. <https://www.enthought.com/blog/neural-network-output-layer>. Accessed: 2025-06-01.
- [26] NATARAJAN, Sriraam; SURESH, Harini; DOMINGOS, Pedro and OTHERS, 2023. Shallow architectures can match deep networks on tabular data. *Scientific Reports*. 2023. 13, num. 1, p. 5634. Available from: 10.1038/s41598-023-32559-8. Available from: <https://www.nature.com/articles/s41598-023-32559-8>.
- [27] KINGMA, Diederik P. and BA, Jimmy, 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*. 2014. Available from: <https://arxiv.org/abs/1412.6980>.

LIST OF FIGURES

Figure 2.1	Distribution of the target variable: Real Effort Person Hours.....	40
Figure 2.2	Boxplot of Real Effort Person Hours by DataDonator	41
Figure 2.3	Correlation Heatmap of Variables.....	42
Figure 2.4	Grid search results for feature selection using linear regression. Configurations include baseline UCP metrics and various decompositions of compound features into their raw components.....	53
Figure 2.5	Feature Selection and model grid search using R^2	69
Figure 2.6	Feature Selection and model grid search using RMSE.....	70
Figure 2.7	Architecture of best performing neural network	81
Figure 3.1	Neural network: Distribution of prediction errors.	84
Figure 3.2	Ridge regression: Distribution of prediction errors.	84
Figure 3.3	Neural network: Histogram of prediction errors with KDE.	85
Figure 3.4	Ridge regression: Histogram of prediction errors with KDE.	85
Figure 3.5	Neural network: Error vs predicted value.	86
Figure 3.6	Ridge regression: Error vs predicted value.....	86
Figure 3.7	Neural network: Predicted vs actual person-hours.	87
Figure 3.8	Ridge regression: Predicted vs actual person-hours.....	87

LIST OF TABLES

Tabulce 2.1	Individual categories in sector are well represented	37
Tabulce 2.2	The programming languages used in the projects and their counts. C# was the most frequently used language. The languages only used once or a handful of times are not useful as predictors.	38
Tabulce 2.3	The methodology used in the projects and their counts. Waterfall was almost exclusively the only methodology used. Therefore, this predictor is not useful	38
Tabulce 2.4	The application types used in the projects and their counts. Business applications dominated the dataset, while other types were underrepresented.....	38
Tabulce 2.5	Outlier Detection Results: IQR Method	44
Tabulce 2.6	Z-Score Outlier Detection Results (threshold = 3)	45
Tabulce 2.7	Three example entries for actor counts before and after scaling.	59
Tabulce 2.8	K-Fold Cross-Validation Results for Ridge Regression.....	74
Tabulce 2.9	Neural Network Grid Search Parameters	78
Tabulce 2.10	K-Fold Cross-Validation Results for Neural Network.....	79
Tabulce 2.11	The configuration of the best-performing neural network	80
Tabulce 3.1	Comparison of K-Fold Cross-Validation Results Between Ridge Regression and Neural Network	82