Empirical Techniques For Effort Estimation in Designing Effective ML Models

, Padmanabha Reddy Y C A[2,7][0000−0002−9697−7072], Sai Sathwik Kosuru[1,5][0009−0008−8057−8799] ⋆, Nageswara Rao Sirisalla[3,8], Vivekananda GN[4,9], and Perala Venkata Akanksha[1,6]

[1] Department of CSE, B V Raju Institute of Technology, Telangana 502313, India
[2] Department of Artificial Intelligence and Machine Learning, Chaitanya Bharati Institute of Technology, Gandipet 500075, India
[3] Department of Computer Science and Engineering, K.S.R.M.college of Engineering, kadapa AP.
[4] School of Computer Science Engineering and Information Systems, Vellore Institute of Technology, Vellore 632014, India
[5] saisathwik363@gmail.com
[6] akankshaperala2003@gmail.com
[7] padmanabhareddy_cseaiml@cbit.ac.in
[8] nagsirisala@gmail.com
[9] vivekananda.gn@vit.ac.in

Abstract. Machine Learning Research often involves the use of diverse libraries, modules, and pseudocodes for data processing, cleaning, filtering, pattern recognition, and computer intelligence. Quantization of Effort Required for the above cumulative processes is rarely discussed in the existing works & The time to reach the desired level of the model's functionality is essential to gauge the environment training for the model training & pre-deployment testing. In this study, we empirically defined the manual-cum-computational effort required for the model development in terms of 2-time factors: time-to-live(time until 1st code modification) and time-for-modification(time between 2 codebase changes) taken together in a mathematical model to compute Effort Factor(quantitative measure of determining effort needed to design ML algorithms). The study is novel in terms of how the effort required to create ML models is calculated with a particular focus on the manual effort direction. The results and the findings obtained can be used for determining the total time needed to synthesize ML models & frameworks in terms of code change cycles and implementation strategies marking 25% performance increase in the current method above the Standard Pipeline.

Keywords: ML Model Design · Code Quality · Effort Factor · Time-To-Live · Time-For-Modification

⋆ Corresponding Author

1 Introduction

Machine Learning models are constructed with modules concerning Feature Extraction, Feature Engineering, Data Filtering, Model Building, Model Testing, Model Deployment, and MLOPs(Machine Learning Operations) implementation. While the computational requirements for such processes are significantly high, the manual inputs and the developer efforts significantly enhance the proportion of effort within the implementation scope of the total model-building time. As is common with every system implementation, Machine Learning Models are built by recursive and iterative stacking of several basic implementation-specific code blocks that collectively produce the desired results. The desired result for a Machine Learning model is the completion of training and suitability for on-demand testing.

Jupyter and Google Colab are the top-most preferred Integrated Development Environments to code ML models owing to their library and infrastructural support. They provide a cell-wise code snippet feature that exploits the interpreted nature of the Python Programming Language to produce line-wise and line-specific outputs and results. The total code implementation stands valid provided the basic syntactical structure of the written code does not affect the Python compiler's operations on making the codes executable. Thanks to the extensive open-source and closed-source library and module support by the IDE, we need not consider the deep mathematical foundations within the model logic and we can just focus on the expected functionality of the developed model.

Software Effort Estimation and analysis in the past broadly encompass the concepts and theories of Industrial Practices, Process Models, and Diverse Design Principles that entail the involvement of significant efforts including capital budgeting, workforce enforcement, Framing regulations, and Consideration for The Management of Large-scale Projects. While LLMs have been used in the existing works for the codebase analysis, Lexical-cum-Syntactical analysis was not performed at a scale similar to the one performed here. There is no substantial evidence of the dynamic nature of the code analysis in the existing systems which was a novel topic referred to in the current study.

Machine Learning Model Training-cum-Testing processes are in themselves computationally and theoretically time-consuming where we do not consider the impact of the codebase versioning. If we include the codebase modifications in place, we may see unusual patterns of effort requirements to code these models which is an interesting aspect worth analyzing. On a higher-level view, The findings[1] here can contribute to/can be used/ broadly applied as a reference:

- By the open-source contributors for the In-Situ Modifications of the open-source Machine Learning Libraries, Modules, and Code Repositories via pull-push workflows.
- For beginners, novice enthusiasts, and students to determine optimal workflows to understand and minimize the time taken in implementing their projects.

[1] Note: The Study provides Generic Findings unspecific to any particular codebase.

– For Companies and Domain Experts to consider improving the developers'/employees' ML Model effort requirements by modifications of architectural parameters.

Table 1: Existing Systems' comparison with the current study

Conceptual Changes	**Description**
[16] [17] Time + Price + Manpower $\longrightarrow$ **Codebase Changes**	Software-only Version
30% $\Longrightarrow$ 80%	
[18] [19] ML Techniques for Effort Estimation $\longrightarrow$ **Effort Determination on ML Codebase**	Effort for Model Building
80% $\Longrightarrow$ 82.3%	
[20] [21] Compiler & Interpreter Perspective (Line-wise analysis) $\longrightarrow$ **Hyperparameter & Training Perspective**	Conventional Software Projects To The ML Model Analysis
65% $\Longrightarrow$ 96.8%	
[22] [23] Syntax & Semantics Tokens considered $\longrightarrow$ **Syntax Trees & Hyperparameter sets**	Architectural Stereotypic Analysis
65% $\Longrightarrow$ 96.8%	

2 Literature Review

2.1 Background Study

Effort etymologically doesn't coerce properly with various definitions provided in technology, particularly the domain of Software Engineering in the Field of Computer Science. In one area, it is an approach to reach the desired accuracy for a trained model [1] while in a broader definition provided by another field, it is a collective term referring to the procedures for planning, managing, and deploying software projects within the required scope& constraints [2]. Reverse Engineering of Machine Learning for Software Effort Estimation is also commonly prevalent across much of the related literature, where machine learning algorithms and techniques are employed to meet the effort requirements based on project constraints and scope. Artificial Neural Networks and Various Fuzzy Logics are often used for Effort-Development Mapping [5] where we estimate the effort measurements based on the much more accurate predictions by the developed ML models and techniques [3]. To support the above assertions, It is worth noting that We have various code repositories [24] [25] and project documentation available on the internet which we can use for training the Machine

Learning Models that demonstrate clear learning rate enhancements, thus providing predictions with higher accuracies [4]. Thanks to the advancements in the computational data infrastructures, we can observe the above implementation at the feature level [3] [4] [5] where we can observe the Machine Learning Models such as ANN, SVM, K*, etc., being performed for deeper analysis and concrete pattern recognition.

Effort Enhancement is quite evident in Machine Learning and Deep Learning owing to the need for precise predictions and improved accuracies in the developed models and hence, the research-oriented analysis was performed towards the same [6] with some works involving even ensemble- and heuristic-based learning approaches [7]. For Statistical and numerical measurement of prediction availabilities for a model, there is some effort laid towards cross-validated choice and assessment [8] with some numerical techniques like linear equation least square analysis [9], Discriminatory Analysis Error Rae Determination [10], Optimal Statistical Decisions [11], etc. More unconventional numerical measures of effort do exist with examples like person-hours, person-months [12], expert-provided software effort [13], Latent Dirichlet Allocation (LDA) [14], Use Case Points (UCP) [15], etc.

2.2 Existing System

Scenarios For The Existing Systems' Implementation

Scenario 1 Software Effort Estimation is broadly directed toward project planning protocols and effort calculations. The effort is a qualitative relation between time, price, and manpower variables.
Scenario 2 Metrics were based on the time taken by developers in designing systems in days and hours. Budget planning and project management concepts apply here.
Scenario 3 Machine Learning Algorithms have been trained on several data sets available containing effort requirements and project effort components.

Differentiators For The Current Work Table 1& 2 details the critical differentiators highlighting the need for separate Software Effort Estimation for the current domain.
Parametric Differences: Time, Price, and Manpower are categorically measured to estimate the effort for designing necessary software applications which is common to the ML domain also. While the current paper does not systematically test these aspects, it analyses the Codebase Corpora to measure developer time and efficiency to produce the desired model and the improvement observed is **+20%**.
Reverse Engineering: Machine Learning Techniques offer detailed analytics on software effort based on the commonality observed among the diverse data sets. We hereby use Software Effort Estimation numerically to research various architectures for optimality & efficiency.
Perspective Interpretation: We perform post-deployment and post-development

analysis of the model functioning considering it is accuracy and learning rate which can succeed in the software effort that performs line-wise analysis on the codebase based on compilation and interpretation operations.
Tokenizers: Syntax trees, Hyperparameter sets employed for analysis where Architectural parameters are also considered. This implies a significant transformation between conventional Software Projects(Syntactic and Semantic Analysis performed here) and Novel ML techniques with impact improvement from 65% to 96.8%.

3 Methodology

3.1 Empirical Terms

Certain Important Elements can be empirically defined as new terms for a complete theoretical description of the current study. We could list 5 terms to empirically design the effort factor defined as follows:
Effort Factor Numerical Quantity that defines the way the ML Models are coded and theoretically modified where we consider the Codebase Adaptations in terms of Convergent and Divergent results where unit selection is dynamic.
Valid State[2] The state where the model behaves expectedly, acting as a template for our desired model, and mirroring positive progress towards building the final model according to our requirements. Associated with Time To Live.
for **Invalid State** [10]The state of chaos where lower accuracies, poor model performances, syntactical and semantic errors, and things that require modifications. Associated with Time For Modification.
[2]**Time To Live**[3] The Time between the coded version and the first modification which is principally the first time interval to build the desired model.
Time For Modification [3] The interval between the consecutive codebase changes where the ML Model being developed transitions between the Valid & Invalid States.

3.2 Mapping The Discrete Divisions Between The Phases

Mapping I Initially, we start with an uncertain state that is devoid of any specification on the validity and veracity of the model development. The processes consist of Data Collection & Data Processing Procedures where the data set is accessed on the following non-exhaustive modes with different Times-To-Live:
CSV With 90% more availability and 99.5% increased access speed, pd.readCSV() results in a probabilistic success rate above 0.855.

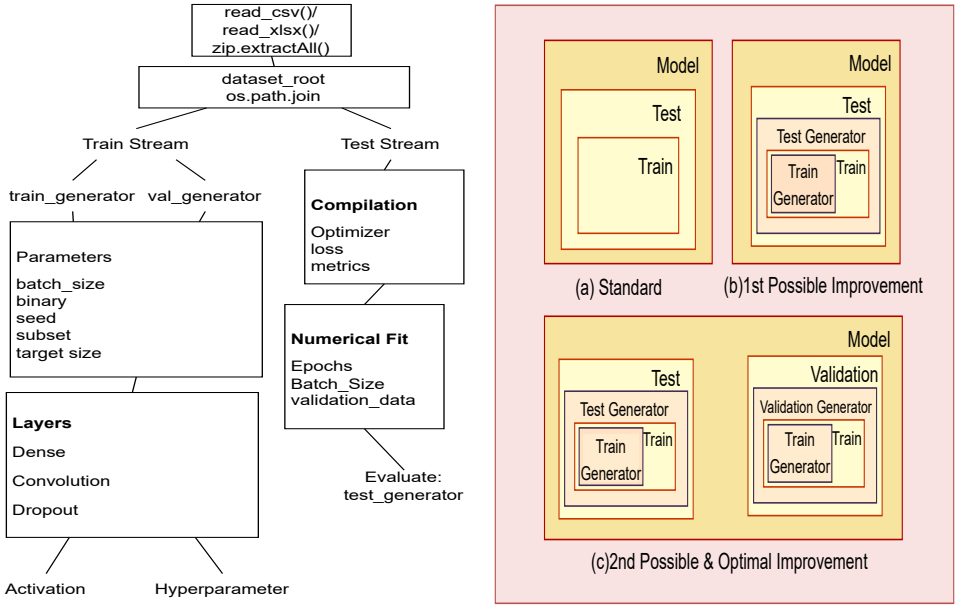
[10] Terms **State** & **Stage** are interchangeably used with **Phase**, wherever appropriate.
[2] Terms **State** & **Stage** are interchangeably used with **Phase**, wherever appropriate.
[3] The interval between the consecutive codebase changes where the ML Model being developed transitions between the Valid & Invalid States.
[3] With the term **cycle**, it indicates iterations for complete transition between phases.

Table 2: Existing Work Comparisons

Author	**Purpose**	**Theory**	**Concept**	**Weaknesses**	**Improvement From This Work**
Cochran (1968) [10]	To model the error rate numerically	Appl.-Based	Error Analysis	Fails to gauge error estimation effort	We also focus on time required to overcome these errors
Jadhav (2023) [11]	To use intelligence systems for estimating software effort	Software Effort	Effort Estimation	Does Not Concentrate on AI Models	Focus on Effort Required For Effort Models
Dantas (2018) [15]	To model effort for Optimal Software	Optimal Software Effort	Optimal Coding Approaches	Predictive Techniques may enhance this	Effort Estimation techniques proposed here enhance the optimality
Ali (2023) [16]	To Create Ensemble For Software Effort Estimation	Model Selection for Effort Estimation	Ensemble Learning For Software Effort	Model Voting may not be diverse	We proposed a collection of numerical techniques to counter this
Tue Nhi (2023) [17]	To show the relevance of AI in effort estimation	Theory of Correlation	Numerical Analysis of Comparison between AI and Software Effort	Theoretical Analysis with a few numerical techniques	We enhanced these numerical equations and proposed novel techniques
Cummin (2021) [21]	To optimize software effort for AI Research	Another Way of Effort Estimation	Theoretical Effort Analysis	Does Not cover deep learning models	We cover the deficiencies found

Zip File This access method is efficient owing to the requirement for larger data sizes in the training processes where the extra overhead of data extraction is added to the total effort factor.

Equations 1, 2 & 3 illustrate the respective theoretical non-numeric relations for the total process. Based on our experimental constraints, we have used



(a) Syntax Tree (b) Synthetic Code Improvements

Fig. 1: Synthetic CodeBase Analysis

zip.extractAll() method instead of pd.readcsv() due to the need for diverse data sizes during the training process. The parameters in the original form are only the filename while it is the zip and the content files compressed for the transformed case. We need to consider a numerical function covering parameters: Time-To-Live, Time-For-Modification, State, and Access Mode for this. It is found that the access mode is the most important factor that determines the net effort in this initial step.

Some modes of file access will still be investigated in future work which is empirically assumed as the following set of equations:

$$pd.readcsv() \longrightarrow \textbf{zip.extractAll()}(\text{Numerical Heuristics Used}) \quad (1)$$

$$\text{f(,,filename)} \rightarrow \textbf{f(zip,content,,)}(\text{Missing Parameters Project-Dependent}) \quad (2)$$

$$\text{effort-factor} = \text{f}(\text{t}_{\text{live}}, \text{t}_{\text{modification}}, \text{state,access mode}) \quad (3)$$

Mapping II Feature Engineering & Extraction(numerically represented in Eq 4) actively adjusts the model parameters between the Valid and Invalid states where Time-To-Live is generally higher compared to other metrics. This is because the behavioral dynamics of ML Models do not resonate well with the definite set of selected features from the data set implying no direct mapping between the parameters. In Equation 4, the point that highlights the relevance of pattern recognition in the process is the transformation from a variable pixelated diagram(Image/Video Frame) into a fixed(256px size in 2D transformation

and 3 color channels for 3D representation are shown here) focusing on the convergent pattern recognition and clustering among the constituent pixels.

$$\text{variable-pixel diagram} \rightarrow \textbf{256px(2D) size \& 3(color) channels} \quad (4)$$

Mapping III Architectural Designs and Hyperparameter tuning significantly influence the predictability power of an ML Model with flexible additions and eliminations of Dropout layers during the process. Equation 5 matches the existential parameters with the Time-To-Modification cycles required for building the desired model.

$$\text{effort-factor} = \text{f}(\text{t}_{\text{modification}}, \text{state}, \text{type}_{\text{layer}}) \quad (5)$$

Mapping IV Deployment Statistics(Table 3) indicate some independent characteristics of Model Saving, Testing, and Predictability. 75% of inspection is performed on Model Testing, 10% on Predictability, and 5% on saving the model. Unfortunately, there exists no fail-proof mechanism for saving the model where there is a risk of losing the training progress of the model.

Table 3: Mapping IV

Testing(Pre Deployment)	Poor Accuracy	Hyperparameter Tuning
Saving(Deployment)	Training Progress Loss	Model Retraining

3.3 Class-Based Approach

We have considered a multi-class data set containing 1,20,000 plant disease images categorized across 10 classes. The Reason for the selection and consideration of the data repository is:

- Leaf Images are broadly dense in pixel arrangements where contour distributions are sufficiently coarse, thus highlighting the complexity of data that the model can deal with. The classification/categorization of the diseases in the data set is not particularly important in the current study.
- The Pattern Recognition task in this data set is generic in its approach facilitating the universality approach of the current work.
- We found very minute effort factor approximations for the data repository with less quantity of classes and the count of 10 classes in the selected data set is reasonably considerable.

Each class demonstrates some depth of Pattern Recognition task complexities, tabulated in Table 4. The following observations from table 4 further justify the relevance of the data set:

Table 4: Class-Layer Relationships(o-optional,m-mandatory layer addition)

No of Classes	I	II	III	IV	V	VI,VII,VIII	IX,X
(layers, dropout)	(3,0)	(3,0)	(3,1(o))	(4,1(m))	(5,1(o))	(10,1(m))	(16-32,3)

- There is no significant difference among 1-3 classes for the number of model layers diluting their importance for model development.
- Overfitting is highly possible for 5-10 classes, essentially complex for the current study.
- Optional dropout doesn't guarantee a minimum number of layers as evident in 5 layers for 5 classes.

As a result, A minimum of 8 classes seems sufficient for the current study where diversity is too important following the number of classes.

3.4 Schema of the Study

Table 5 shows a vivid representation of the standard pathways between valid and invalid states, involving Time-To-Live and Time-For-Modification parametric features. We systematically divide the components into different inspection phases involving the different stages along a Model Development Pipeline demonstrated in Figure 2. **Standard CodeBase Synthesis** The overall code of the

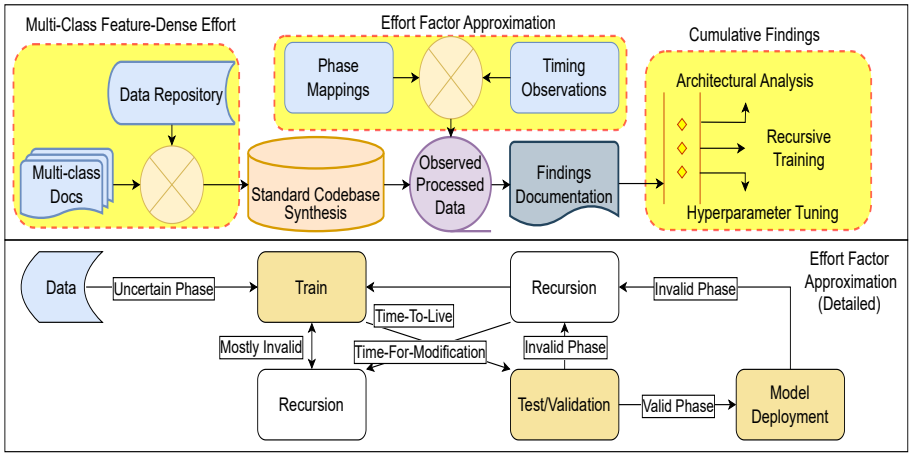


Fig. 2: Process Flow in the Study

ML Model and the Required Data Pipeline is coded, popularly in Python(preference due to Diverse Library Support). The Syntactic elements and the Syntax Trees

Table 5: Theoretical Schema of Operations

Valid Phase	Invalid Phase
Standard CodeBase Synthesis Compilation Execution	
	Structural Fault in the Model Architectural Fault Hyperparameter Defect
	Codebase Inspection Time To Live Time For Modification
Model Inspection *+Time To Live* Accuracy Estimation Learning Rate Determination	
	Feature Inspection *+Time For Modification* Effort Factor Accuracy Estimation
Deployment Testing *Compile & Predict* Save Model Test on Data Set *-Time To Live*	
	Recursive Inspection *-Time For Modification* Architectural Modification Data Set Modification Synthetic Data Implementation Retraining Model

incorporated in this phase are shown in Figure 1a and the way the code is improved step-by-step is demonstrated in Figure 1b. Compilation-cum-Execution produces the necessary transition between the states in the subsequent steps.

Structural Fault in the Model, if encountered, may be due to Architectural Fault or Hyperparameter defects. The Number of Layers can significantly affect the model's behavioral aspects, learning rates, and learning patterns. This invalid state needs a detailed **CodeBase Inspection** for any corrections related to the code that may positively impact the model's performance. **Model Inspection** is performed by modelname.predict() method where the accuracy computed is recorded and the learning rate is estimated. All of these add to the Time To Live and then, **Feature Inspection** is performed which is subsequently added to the Time For Modification. Here, we initiate efforts to change codes

and bring in the desired model and the accuracy is recursively estimated till we get near-expected values. On Deployment, the model undergoes **Deployment Testing**: Compilation and prediction cycles where the model is saved and tested on the Data Set and the Time To Live is reset. From here, recursive cycles of the above happen where Time For Modification is reset before initiation of the stage with Architectural, Data Set Modifications, and Synthetic Data Generation & Model Retraining procedures as the component processes.

4 Results and Discussions

4.1 Observation-Based Analytical Aspects

Phase-Based Transition Empirical Analytics All the observed transitions between the Valid & Invalid Phases with simultaneous variations in Time-To-Live & Time-For-Modification have been mapped in figures 3a & 3b, the obtained results are compared to the standard ML Pipeline used in existing works in table 8 and the whole process is already demonstrated in table 5 and 6.
i. Valid Phase: The development dynamics of the model building are found to be too stable at about 10% learning ability retention for the process to incline more towards the active phase with 0.824 probability of success than the inactive phase though it originally tends to be neutral in terms of effort requirements before the training phase. We have observed two ways in each phase, which we term entry and exit:
Entry Data is collected according to the requirements which accounts for 60%

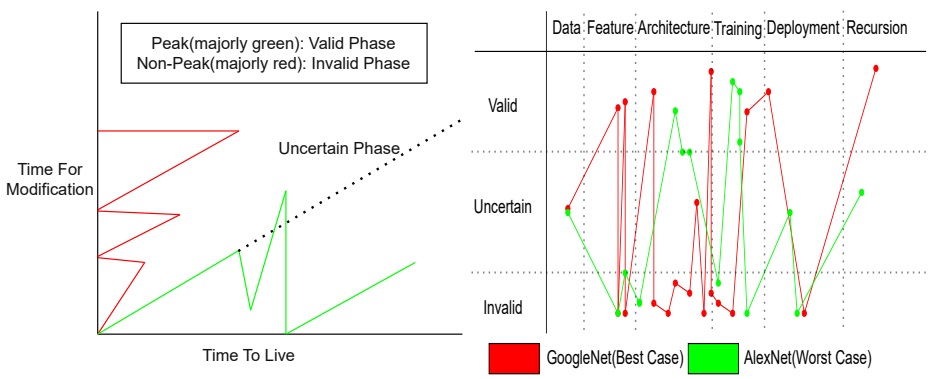
Table 6: Entry-Exit Relational Results in our Study

Phase	Step	P(Entry)	P(Exit)	$ComparisonRatio_{effortfactor}$
Valid	Data Processing	0.625	0.213	9:2
	Feature Extraction	0.723	0.113	5:1
	Model Building	0.235	0.935	4:3.5
	Model Training	0.552	0.502	1:1
	Model Testing	0.832	0.222	9:1
InValid	Data Processing	0.225	0.413	1:1
	Feature Extraction	0.223	0.553	2:5
	Model Building	0.135	0.924	3:4
	Model Training	0.112	0.602	1:1
	Model Testing	0.132	0.922	2:1

of the effort at this point, and processed through feature extraction occupying 20% of the effort approximation, feature engineering, dimensionality reduction spanning a meager 0.5-0.8% of the total effort applied, and feature selection. A

strong tendency to remain stuck to the requirements of the model's desired functionality causes 82.3% of the initial phase of the development cycle to become the active phase. This marks a significant and considerable entry into the active phase. Model Inspection, with an additional Time To Live, revolves around Accuracy Estimation and the Learning Rate Determination(provided the required accuracy is obtained at this stage. The last entry into this phase is recorded at the Deployment where the model code is compiled & predicted accounting for 2% contribution, followed by model saving and testing on the data set with an equivalent depreciation in the Time To Live. Model saving and testing have a very diminished presence in the effort manipulation in this phase.

Exit Exit is primarily constituted with the Time For Modification cycles(around



(a) Phase Transitions across timings (b) Worst & Best Case Findings

Fig. 3: Relation Between The Phase Transitions And The Timing Findings

3-4) with recursive loops, whenever necessary. In the event of Hyperparameter Defects and Architectural Faults, the noticeable re-entry is not noticed until the model inspection phase that follows next. Lower accuracy of <0.3 and Poor Performance also cause sufficient exit conditions to this phase where we see modification phases like Feature Inspection and Recursive Inspection for re-entry. Improved Accuracy and desired behavior are the critical factors that determine the efficiency of re-entry. The time for modification has no relevance concerning the entry and exit processes in the valid phase. The relevant equation is shown in the equation 6.

$$f(\text{effortvalid})+f(\text{effortinvalid}) \rightarrow p(t_{\text{valid}})*p(\text{non-error})+p(t_{\text{invalid}})*p(\text{error})+p(\text{re-entry}) \tag{6}$$

ii. Invalid Phase: Invalid Phase steps are characterized by sufficiently different modification and rectification cycles marked with diverse reviews and iterations.

Entry The First entry into the invalid phase is observed when Component-

based structure faults(accounts to effort contribution to about 25%) are noticed in the codebase. In Codebase Inspection, followed by Model Inspection, with the faulty inspection being the intermediate procedure, we can notice the relevance of Feature Inspection involving the computation of the Effort Factor(contribution of 70%) and Accuracy Estimation(contribution of 30%). A significantly time-consuming phase succeeds this phase where we check the deployment model where unexpected model behavior with a contribution of 40%, Lower Accuracy with a contribution of 25%, and Poor Performance with a cumulative contribution of 70% cause re-entry into this phase with looping Time For Modification increments. Here, Recursive calls into Architectural(contribution: 80%), & Data Set(contribution: 20%) Modification, including Synthetic Data Implementation and Retraining Model, take place, marking significant iterative re-entries into the invalid phase with simultaneous Time For Modification value depreciation.

Exit The Exit from the Invalid Phase is comparatively simple when compared to the Valid Phase. Improvement in Performance, Alignment of the Model Behavior with the Line of Expected End Model, and Identification of the Correct Architectural Combination facilitate the smooth exit from this phase. The relation is demonstrated in the equation 7.

$$f(\text{time-in-invalid}) = \text{t}_{codebaseinspection} + \text{t}_{featureengineering} + \text{t}_{performanceassessment} \tag{7}$$

Empirical Validation: While our theoretical framework provides insights into phase transitions, empirical studies are crucial to validate these observations rigorously. Future studies could focus on:

1. Longitudinal Case Studies: Temporal tracking of model architectural evolution & performance transitions.
2. Quantitative Metrics: Develop metrics to quantify the transition dynamics between Valid and Invalid Phases, such as frequency, duration, and impact on architectural parameters.
3. Comparative Analyses: Comparing our proposed model development phases with existent studies from diverse machine learning projects to identify common patterns and deviations.
4. Simulation and Modeling: Using simulation techniques to model different scenarios of phase transitions and validate theoretical predictions against simulated data.

These empirical approaches would provide a more robust understanding of the dynamics described and validate the applicability of our phase-based analytical framework in practical machine-learning settings.

Timing Findings The findings on the time factors involved are substantially concrete and can be easily tabulated as in Table 7 and were detailed in the following section.

Effort Factor Approximation The maximum computed Time To Live value is at 6000 seconds while that for Time For Modification is 300 seconds and there

Table 7: Timings Findings with % Contributions To The Total Obtained Timing Result For Our Experiment

Time-To-Live Findings		**Time-For-Modification Findings**	
Experiment	%Time	Experiment	%Time
Accuracy > 0.5	92.2%	Architectural Fault	92.2%
Desired Accuracy(If Any)	60%	Wrong Combination of Hyperparameters	20%
Higher Learning Rate	50%	Effort Factor	99.8%
Lower Learning Rate	25%	Accuracy Estimation	99.8%
Compile & Predict	55.2%	Synthetic Data Implementation & Retraining	5%

is no minimum numerical value for Time To Live whereas the Minimum value observed for Time For Modification is 100ms. The effort is composed of the following immediate changes with % contribution to the total timing findings:

1. Before the Time For Modification lapse:
   - Architectural Layer Modifications - 72%
   - Hyperparameter Tuning - 23%
   - Dropout for Overfitting prevention - 3%
   - Regularization and Miscellaneous Factors - 2%
2. Before the Time To Live lapse:
   - Hyperparameter Tuning - 99.5%
   - Addition of More Regularization & Dropout Parameters - 0.5%

The inferences regarding the above observations for the effort factor determination include:

- Initial Running of ML algorithms contains only verification checks for the Architecture and Hyperparameter combinations.
- Corrections include Architecture Modifications & Hyperparameter tuning as Mandatory steps and Dropout & Regularization parameters optionally.

Equations 3,4,5 are further found to expand into

$$\text{effort-factor} = \frac{\text{code change heuristic} \times \text{\% impact}}{\text{total code changes} \times \text{total model impact}} + \text{f}(\text{t}_{\text{live}}, \text{t}_{\text{modification}}) \tag{8}$$

$$\text{effort-factor} = (\text{pixel\_size}_{\text{set}} - \text{pixel\_size}_{\text{original}}) \times \text{color-metric} + \text{oth. heuristics} \tag{9}$$

$$\text{effort-factor} = \text{f}(\text{state}, \text{f'}(\text{type}_{\text{layer}}, \text{hyperparameter}_{\text{layer}})) + \delta \text{t}_{\text{modification}} \tag{10}$$

Equation 10 demonstrates the complex nature of architectural modifications required for retaining the valid state throughout the process. The involvement of the composite function f' with the parameters *type of the layer* and *hyperparameter set* $\{h_1, h_2, \ldots, h_n\}$, where $h_1, \ldots, h_n$ is the set of hyperparameters for

the layer, highlighting the need for further research on this aspect. The functions t_{live} and $t_{\text{modification}}$ in Equation 8 are dynamic with the project's specific requirements, and there are no generic criteria to expand the function. Other heuristics in Equation 9 are also dependent on a project's specific requirements and need to be decided by the respective project manager. It is noteworthy to mention that The difference in rescaling the figures of the data set has a reflective impact on all the constituent color channels(Equation 9) and Respective Heuristics assigned for code changes in Equation 8 are based on the complexity of the neighboring code snippets, hence reflecting the local scope of the **Standard CodeBase Synthesis**.

4.2 Post-Analytic Review of the Findings

Table 8: Effort Comparisons between Normal(Oth.)&Proposed(Ours) Pipelines

Iterations ↓	Train		Test		Train		Test	
Pipelines →	Oth.	Ours	Oth.	Ours	Oth.	Ours	Oth.	Ours
Parameters →	Total Effort (hr)				Time-To-Live (min)			
I	10hr	**6hr**	1hr	**0.75hr**	15min	**5min**	5min	**0.5min**
II	8hr	**6.2hr**	1hr	**0.72hr**	22min	**7min**	3min	**0.5min**
III	8.2hr	**5hr**	0.5hr	**0.4hr**	5min	**5min**	1.5min	3.2min
IV	9hr	**7.5hr**	0.2hr	**0.18hr**	32min	**10min**	15min	**2min**
Parameters →	Time-For-Modification (min)				Effort(in %) In Invalid Phase			
I	4min	**8min**	5.2min	**2min**	50%	**85%**	95%	**99.5%**
II	1min	2.2min	3.2min	**1.1min**	65%	**92%**	98%	98%
III	6min	**2.1min**	2.1min	**1min**	52%	**75%**	98.5%	**99.2%**
IV	4min	**2min**	1.2min	**0.5min**	82%	**91.2%**	97.3%	**99.2%**

To conclude on the results in the Table 8, we need to note 3 critical points:

1. Time To Live and Time For Modification are too minute in values to calculate them in seconds(the majority of them are in *ms*), hence percentages were considered.
2. Valid and Invalid phases' line of separation is too wide such that there are fewer commonalities among them.
3. Entries and Exit explanations are purely theoretical calling for Detailed Analytics of the Timings Findings only.

From Table 7 and 9, we can infer that certain theoretical aspects significantly impact the timing results. While the validity changes dynamically, we can isolate the timing findings for sufficient insights into the analytical results as follows.

Time To Live Findings An optimal accuracy computation generally takes less effort than a standard effort to obtain the desired accuracy, if it is provided by the user. We could observe a significant difference of **32.2%** difference between the corresponding timing proportions. Higher Learning Rate & Lower Learning Rate have mixed contributions to the required effort calculations with values of **50%** & **25%**. Compile & Predict negatively affect the Time To Live where we save effort in further improving our requirements and model performances where we could see the percentile contribution of **55.2%**. It has the following inferences:

- Time To Live is a manifestation of the residence time of the model development in the valid phase where better accuracy guarantees a greater amount.
- The proportion of the model development in the valid phase is proportionately less highlighting the complexity of the problem.
- Learning Rate, Accuracy, Architectural Parameters facilitate transitions between the valid & invalid stages thus affecting the Time To Live value variations.

Table 9: Time To Live& Time To Modification Comparison Results in our Study

Phase	Step	P(t_{live})	P($t_{modification}$)	$ComparisonRatio_{timings}$
Valid	Data Processing	9.1s	10.2s	9:4
	Feature Extraction	15.4s	10.2s	8:7
	Model Building	2.3s	2.3s	6:3.2
	Model Training	80.5s	92.2s	1:1
	Model Testing	1.2s	2.2s	1:2
InValid	Data Processing	2.2s	1.8s	1:1
	Feature Extraction	4.5s	5.5s	2:5
	Model Building	10.9s	15.4s	3:4
	Model Training	100.2s	75.5s	1:1
	Model Testing	1.2s	0.92s	2:1

Time For Modification Findings Efficient Hyperparameter Tuning ensures the necessary architectural patterns required for a much more accurate ML/DL model thus enhancing the Time For Modification in the initial stages of the Model Development only. Here, we have a 2-pronged problem with **92.2%** Architectural Fault & **20%** Wrong Combination of Hyperparameters. Effort Factor and Accuracy Estimation each make the same contribution to the Time For Modification by a magnitude of **99.8%**. Synthetic Data Implementation is also employed in certain cases where the size of the data set considered is small and Model Retraining where any process leading to the required Model's development annuls the overall aim of the whole process with each contributing **5%**.

The inferences can be:

- Greater Portion of the Model Development is covered with the Invalid Phase with Time For Modification increments along the process.
- The process of the Invalid Stage is highly iterative & recursive with repeated architectural parametric changes & hyperparameter tuning with sequential Time For Modification increments & decrements.
- Effort Factor is the major contributor in this stage which helps us determine the percentage of modifications done on the ML code in the Invalid Stage to make it valid.

4.3 Group Comparisons and Probable Efficiencies of the Study

The results obtained so far may be regarded as probability estimations, rather than deterministic mathematical models. While we could model the effort almost exactly, the results are still probabilistic and need further determinism. This leads to group comparisons as the ultimate step in the study, with the results as follows: We have clustered the considered models into three and compared results numerically with graphical representation as seen in figure 4. The clusters are:

Cluster 1 This cluster has Bayesian property as seen in Equation 11 in its effort estimation as modeled in the following equation:

$$\text{p(effort|layered)} = \frac{\text{P}(3 \times \text{layer\_quantity}) \times \text{P}(\text{weight\_factor} \times 0.5 \times \text{effort\_factor})}{\text{P}(3 \times \text{layer\_quantity})} \tag{11}$$

Cluster 2 The cluster has neural networks as its study components which enable probability simulation of each of its nodal compositions. This involves Poisson Distribution Techniques as demonstrated in the equation 12.

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{12}$$

where $\lambda = nf(\text{layers})$ and $\sigma = n^2 f(5 \times 3 \times \text{layers})$

Cluster 3 The cluster involves an intricate analysis of effort for adding additional judgement layers. This is represented by Normal Distribution(Equation 13) as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{13}$$

where $\mu = n_{\text{jdgmnt}}$ and $\sigma = n^2_{\text{jdgmnt}}$

The Group Comparisons are observed as seen in Table 10:

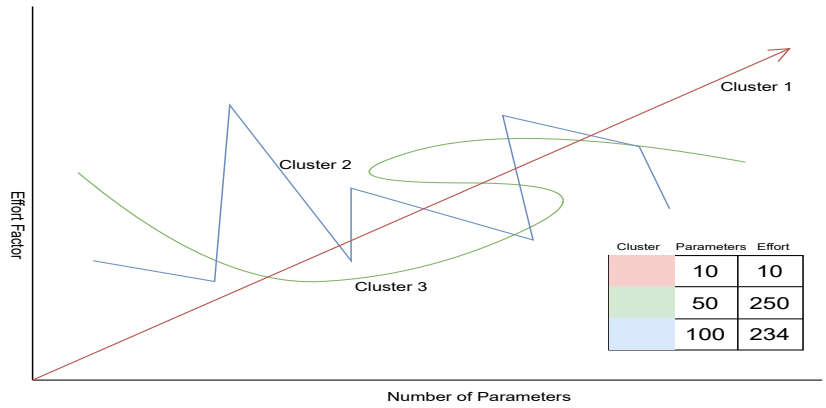


Fig. 4: Graphical Representation of Clustered Group Comparisons

Table 10: Group Comparisons across Study Clusters

Cluster	Step	Others.	Ours.	% Improvement in Performance
Cluster 1	Code Base Analysis	4.5	9.6	51.2%
	Feature Selection	2.1	8.2	79.1%
	Model Application	4.1	9.6	52.9%
Cluster 2	Code Base Analysis	1.2	9.6	81.5%
	Feature Selection	2.1	9.96	81.5%
	Model Application	1.2	4.8	31.2%
Cluster 3	Code Base Analysis	0.1	5.2	49.1%
	Feature Selection	2.1	9.96	61.27%
	Model Application	1.2	8.2	71.1%

5 Conclusion and Future Work

Effort Factor Estimation can greatly enhance our urge to optimize ML model development procedures. With a significant leap from pure accuracy-, performance- & functioning- based analysis of diverse models in the field of data-driven research, this is a unique way of measuring human developers' efforts in making Machines learn from the training data effectively. The study was found to broadly encompass the pre-deployment, deployment, and post-deployment aspects of model development. Having seen some interesting results and variations with different phases like valid & invalid, timing factors like Time-To-Live & Time-For-Modification, we can observe crucial steps that determine whether a process is correctly leading to the expected result i.e., the desired model with considerable accuracy. The future work particularly relies on the efficacy with which the proposed results in this work can transform our approach to Model

Behavioral dynamics. As a part of that, we like to focus on extending the current work to bring in Numerical Estimations of the various observable architectural patterns within these models, including Neural Networks and their impact on the total effort during and final accuracy after the model development process. Further, Experimentally Expanding the Numerical Aspects of The Empirical Equations proposed here is a pending task. Additional Experiments conducted alongside this & Considerations for further research are provided supplemental.

6 Funding Details and Disclosure Statement

The authors report there are no competing interests to declare.

7 Data Availability Statement

There is no data associated with this study since the study is universal.

References

1. Singh, A. J. and Kumar, Mukesh, Comparative Analysis on Prediction of Software Effort Estimation Using Machine Learning Techniques (April 1, 2020). Proceedings of the International Conference on Innovative Computing & Communications (ICICC) 2020, Available at SSRN: https://ssrn.com/abstract=3565813
2. Mahmood, Y., Kama, N., Azmi, A., Khan, A. S., & Ali, M. (2021). Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation. Software: Practice and Experience, 52(1), 39–65. https://doi.org/10.1002/spe.3009
3. Uc-Cetina, V. (2023). Recent Advances in Software Effort Estimation using Machine Learning. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2303.03482
4. Monika and O. P. Sangwan, "Software effort estimation using machine learning techniques," 2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence, Noida, India, 2017, pp. 92-98, doi: 10.1109/CONFLUENCE.2017.7943130.
5. M. Hammad and A. Alqaddoumi, "Features-Level Software Effort Estimation Using Machine Learning Algorithms," 2018 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), Sakhier, Bahrain, 2018, pp. 1-3, doi: 10.1109/3ICT.2018.8855752.
6. Sakhrawi, Z., Sellami, A. & Bouassida, N. Software enhancement effort estimation using correlation-based feature selection and stacking ensemble method. Cluster Comput 25, 2779–2792 (2022). https://doi.org/10.1007/s10586-021-03447-5
7. Rhmann, W., Pandey, B. & Ansari, G.A. Software effort estimation using ensemble of hybrid search-based algorithms based on metaheuristic algorithms. Innovations Syst Softw Eng 18, 309–319 (2022). https://doi.org/10.1007/s11334-020-00377-0
8. Stone, M. (1974). Cross-Validatory choice and assessment of statistical predictions. Journal of the Royal Statistical Society Series B-methodological, 36(2), 111–133. https://doi.org/10.1111/j.2517-6161.1974.tb00994.x

9. Anscombe, F. J. (1967). Topics in the investigation of linear relations fitted by the method of least squares. J. R. Statist. Soc. B, 29, 1–52.
10. Cochran, W. G. (1968). Commentary on estimation of error rates in discriminant analysis. Technometrics, 10, 204–205.
11. Degroot, M. H. (1970). Optimal Statistical Decisions. New York: McGraw-Hill.
12. Jadhav, A., & Shandilya, S. K. (2023). Reliable machine learning models for estimating effective software development efforts: A comparative analysis. Journal of Engineering Research. https://doi.org/10.1016/j.jer.2023.100150
13. M. Haris, F. -F. Chua and A. H. -L. Lim, "An Ensemble-Based Framework to Estimate Software Project Effort," 2023 IEEE 8th International Conference On Software Engineering and Computer Systems (ICSECS), Penang, Malaysia, 2023, pp. 47-52, doi: 10.1109/ICSECS58457.2023.10256337.
14. Sehra, Sumeet Kaur & Brar, Yadwinder & Kaur, Navdeep & Sehra, Sukhjit. (2017). Research Patterns and Trends in Software Effort Estimation. Information and Software Technology. 91. 10.1016/j.infsof.2017.06.002.
15. Dantas, Emanuel & Perkusich, Mirko & Dilorenzo, Ednaldo & Santos, Danilo & Almeida, Hyggo & Perkusich, Angelo. (2018). Effort Estimation in Agile Software Development: an Updated Review. 10.18293/SEKE2018-003.
16. Ali, S. S., Ren, J., Zhang, K., Wu, J., & Liu, C. Heterogeneous Ensemble Model to Optimize Software Effort Estimation Accuracy. IEEE Access, 11:27759-27792, 2023.
17. Tue Nhi Tran, Huyen Tan Tran, & Quy Nam Nguyen. (2023). Leveraging AI for Enhanced Software Effort Estimation: A Comprehensive Study and Framework Proposal. https://doi.org/10.1109/iccd59681.2023.10420603
18. Ayesha Saeed, Wasi Haider Butt, Farwa Kazmi, and Madeha Arif. 2018. Survey of Software Development Effort Estimation Techniques. In Proceedings of the 2018 7th International Conference on Software and Computer Applications (ICSCA '18). Association for Computing Machinery, New York, NY, USA, 82–86. https://doi.org/10.1145/3185089.3185140
19. Shah, R., Shah, V., Nair, A. R., Vyas, T., Desai, S., & Sheshang Degadwala. (2022). Software Effort Estimation using Machine Learning Algorithms. 2022 6th International Conference on Electronics, Communication and Aerospace Technology. https://doi.org/10.1109/iceca55336.2022.10009346
20. H. Leather and C. Cummins, "Machine Learning in Compilers: Past, Present and Future," 2020 Forum for Specification and Design Languages (FDL), Kiel, Germany, 2020, pp. 1-8, doi: 10.1109/FDL50818.2020.9232934.
21. Cummins C, Wasti B, Guo J, et al. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. arXiv. 2021. Available from: https://arxiv.org/abs/2109.08267
22. Gavranović B, Lessard P, Dudzik A, von Glehn T, Araújo JGM, Veličković P. Position: Categorical Deep Learning is an Algebraic Theory of All Architectures. arXiv. 2024.
23. Tao C, Ibrayev T, Roy K. Towards Image Semantics and Syntax Sequence Learning. arXiv. 2024
24. Cuzzocrea, Alfredo. (2023). Advanced Machine Learning Structures over Big Data Repositories: Definitions, Models, Properties, Algorithms. 10.3233/FAIA231000.
25. Su B, Zhang J, Collina N, et al. Analysis of the ICML 2023 Ranking Data: Can Authors' Opinions of Their Own Papers Assist Peer Review in Machine Learning? arXiv. 2024. Available from: https://arxiv.org/abs/2408.13430