Titanic Survival Prediction Using Logistic Regression

(Minimally Preprocessed vs. Fully Preprocessed)

Krishnaswaroop Varati

Robert Smith

EECE 529X

Dr. Xiaohua (Edward) Li

5/12/2025

Abstract

This study aims to evaluate the effectiveness and importance of preprocessing data before using it to train a machine learning model. The data set used was taken from Kaggle.com and is known as the Titanic competition. From this two logistic regression models were trained - one with minimal preprocessing and the other with extensive preprocessing. By using different feature engineering techniques, the objective of comparing classification accuracies helped to understand how preprocessing influences model learning and generalization.

Introduction

Recently, machine learning has become an essential tool for solving classification problems across various industries. A common model that is used for binary classification tasks is a logistic regression model. This is due to its simplicity, efficiency and interpretability. However, this project aims to highlight that the effectiveness of the model is not solely based on the algorithm, but also on how the input data is processed prior to training.

The Titanic dataset from Kaggle.com provides a classic example of a binary classification problem: predicting whether a passenger survived the Titanic disaster based on available features such as pseudo economic class, name, age, sex, number of family members, ticket/cabin number, and which port a passenger embarked from. While the dataset is relatively structured, the quality and format of the features vary. This leads to the need of preprocessing that identifies meaningful patterns.

In this experiment preprocessing takes on many different forms, ranging from filling missing values to creating new variables through feature engineering. In doing so, there can be significant improvement in model performance by improving convergence and boosting the representation of underlying correlations.

Related Work

Data preprocessing is widely recognized as a critical step in the machine learning pipeline. Numerous studies have shown that the quality and structure of input features significantly affect model performance, particularly for linear models such as logistic regression. In the Titanic Kaggle competition, the top submissions highlight the importance of preprocessing, and inspired much of the project. For example, some participants combined the "SibSp" and "Parch" columns into a single "FamilySize" feature and extracted titles from the Name column. In order to use more information, categorical variables such as Sex and Embarked were one hot encoded. Based on certain submissions it can be seen that these enhancements improved classification accuracy by 2-4%

Network Architecture

In this project, "network" refers to the end-to-end machine-learning pipeline that transforms raw Titanic data into survival predictions via a logistic-regression classifier.

Model Formulation

We employ a logistic-regression model to predict the binary outcome $y \in \{0,1\}$ (did not survive vs. survived). Each passenger is represented by a feature vector $x \in \Re^d$, and the model computes -

$$z = w_0 + \sum_{j=1}^{d} w_j x_j \;=\; \mathbf{w}^{\mathrm{T}}\tilde{\mathbf{x}}$$

Where $\bar{x} = [1, x_1, x_2, x_3 \ldots, x_d]^T$ includes a bias term and $w = [w_0, w_1, w_2, w_3 \ldots, w_d]^T$ is the weight vector. The network's output is the sigmoid activation -

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Which estimates the probability of survival.

Loss Function

Training minimizes the binary cross-entropy loss over *n* samples -

$$L(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\Big[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\Big]$$

Where $\hat{y}$ is the output of the model. This convex objective penalizes confident misclassifications heavily, driving the model toward calibrated probability estimates.

Optimization Strategy

We apply batch gradient descent with a fixed learning rate α and a predetermined number of iterations *T*. Gradients ∇ *L(w)* are computed automatically via an auto-differentiation engine. Weight updates follow

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} \;-\; \alpha \nabla L\big(\mathbf{w}^{(t)}\big), \quad t = 0, \ldots, T-1.$$

where w(0) is initialized randomly with small Gaussian noise. Two sets of hyperparameters were tuned based on observed convergence curves -

1. Minimal Preprocessed Pipeline : $\alpha = 0.003$ and $T = 60{,}000$
2. Full Preprocessed Pipeline : $\alpha = 0.01$ and $T = 60{,}000$

The training progress was monitored and displayed as a plot of training loss vs. iterations.

Preprocessing Pipelines

To isolate the impact of feature engineering, we built two distinct pipelines that feed into the same model architecture -

1. Minimal Preprocessing
    a. Encoding : Map "Sex" to { 0 , 1 }
    b. Imputation : Replace missing "Age" values with a statistical fit (median)
    c. Dropping : Remove parameters with minimal effect to training model - "Name", "Ticket", "Cabin", "Embarked"
    d. Resulting Feature Set - { Pclass, Sex, Age, SibSp, Parch, Fare }
2. Full Preprocessing
    a. One-Hot Encoding : Implemented on "Embarked"
    b. Feature Engineering
        i. FamilySize = "SibSp" + "Parch"
        ii. Extract "Title" from name
        iii. CabinDeck - Used just first letter
        iv. TicketType - Numeric vs. Alphanumeric
    c. Imputation : Filled both "Age" and "Fare" with median values
    d. Scaling : normalize continuous features (Age, Fare, FamilySize) to zero mean and unit variance.
    e. Dimension expansion : dummy-var encoding increases the feature count from a handful to dozens, exposing subtle correlations

Both pipelines share the same downstream model code and training regimen, ensuring that observed performance differences stem solely from preprocessing choices.

Software Tools and Implementation

➔ *Data handling* : pandas for CSV I/O and DataFrame operations.

➔ *Numerical computation* : NumPy (via an Autograd-compatible fork) for vectorized linear algebra.

➔ *Auto-differentiation* : Autograd computes exact gradients of *L(w)* with respect to all weights.

➔ *Visualization* : Matplotlib for loss-curve plotting; additional libraries (e.g. Seaborn) may be used for exploratory analyses.

➔ *Environment* : Python 3.x, with the above libraries orchestrated in Jupyter notebooks and standalone scripts.

Training and Testing Strategy

➔ The original Kaggle “train.csv” is divided into a train and test data set (80/20 split) using the “Survived” label

➔ After training, the final weights are applied to the held-out set to compute test accuracy at a decision threshold of 0.5

➔ Report both training accuracy (to verify convergence) and test accuracy (to measure generalization)

This architecture component establishes the foundation for a fair assessment of feature-engineering efficacy by clearly separating preprocessing from model training and closely controlling for hyperparameters and training strategy.

Experimental Results

The classification accuracy for the training datasets were 80.02% (for minimal preprocessed pipeline) and 83.72% (for fully preprocessed pipeline). The following table represents the classification accuracies achieved on the testing dataset.

Pipeline Variant	Classification Accuracy
Minimal Preprocessing	75.8%
Full Preprocessing	77.3%

The minimal pipeline variant showed quick convergence but had a limited feature scope achieving a classification accuracy of 75.8% , whereas the fully preprocessed pipeline had a much richer feature set (FamilySize, Title, Cabin-deck, ticket info, one-hots, scaled Fare) and showed a +1.5% classification accuracy gain.

The following two plots show the training losses for the two pipelines -

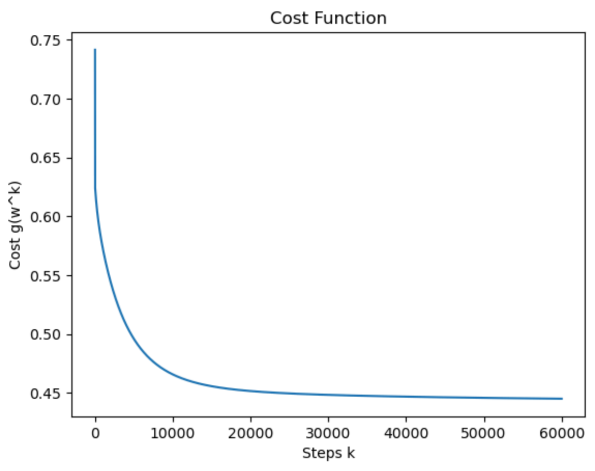

Fig1. Cost function of Minimally preprocessed training dataset

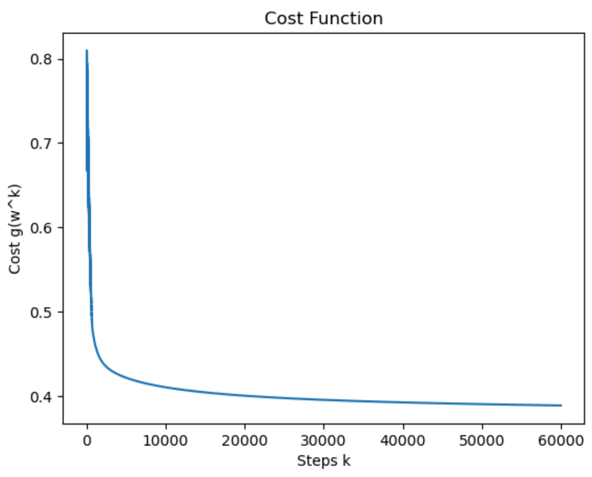

Fig2. Cost function of fully preprocessed training dataset

Conclusion

1. Even a simple feature addition (FamilySize, Title, Cabin-deck) can yield a 1–2 classification accuracy boost in logistic regression.

2. There is a clear trade-off between preprocessing effort and marginal performance gain.

3. For inherently linear models, making domain structure explicit in features often outperforms blind hyperparameter tuning.

4. Future work: compare against regularized or tree-based models, try KNN imputation, or explore polynomial feature interactions.

References

1. Kaggle Titanic competition. https://www.kaggle.com/c/titanic
2. McKinney W. "Data Structures for Statistical Computing in Python," Proc. 9th PyData (2010).
3. Lecture Sets - "EECE 529X : Machine Learning", Dr. Xiaohua (Edward) Li