Introduction to Machine Learning Final Projects
Titanic: Machine Learning from Disaster

鍾承翰
112550189
Contributions:
Code Writing, Report 34%

陳景寬
112550112
Contributions:
Slides, Video Recording 33%

林昌岳
113511248
Contributions:
Slides, Video Recording 33%

Abstract

We investigate feature engineering strategies for Titanic survival prediction through systematic comparison of three pipelines: RF (community practices), XGB (gradient boosting optimized), and MLP (neural network ready). Testing seven algorithms across multiple random seeds, we find that thoughtful feature construction—particularly title extraction and family size derivation—matters more than model complexity. Our Random Forest with RF features achieves 79.67% accuracy, ranking top 8% on Kaggle. Results demonstrate that domain-informed feature engineering remains crucial for small tabular datasets, where tree-based methods outperform neural networks without extensive tuning.

1. Introduction

1.1. Problem Background

The sinking of the RMS Titanic on its maiden voyage represents not only a historical tragedy but also a rich data source for understanding survival patterns under extreme circumstances. The Kaggle competition "Titanic: Machine Learning from Disaster" challenges participants to predict passenger survival based on demographic and ticketing information. With 891 training samples and 418 test samples, each described by features including passenger class (Pclass), name, sex, age, number of siblings/spouses aboard (SibSp), number of parents/children aboard (Parch), ticket number, fare, cabin, and port of embarkation (Embarked), the dataset presents a classic binary classification task with realistic complications: missing values, categorical variables, and potential non-linear interactions between features.

The survival rate in the training set is approximately 38%, indicating moderate class imbalance. Initial exploratory analysis reveals strong predictive signals: women and children had significantly higher survival rates due to the "women and children first" evacuation protocol, first-class passengers enjoyed better access to lifeboats, and fare prices correlated with both passenger class and survival likelihood. However, raw features alone provide insufficient discriminative power—sophisticated feature engineering is required to extract latent information encoded in passenger names (social titles), ticket numbers (group bookings), and cabin assignments (deck locations).

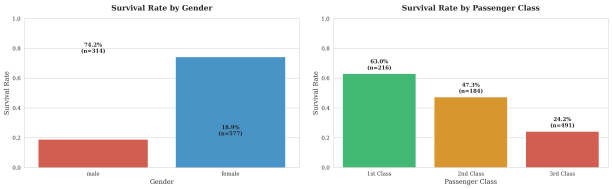

Figure 1. Exploratory analysis of survival patterns: (left) survival rate by gender showing strong bias toward female passengers (74.2% vs 18.9%); (right) survival rate by passenger class demonstrating socioeconomic advantage (63.0% first class vs 24.2% third class).

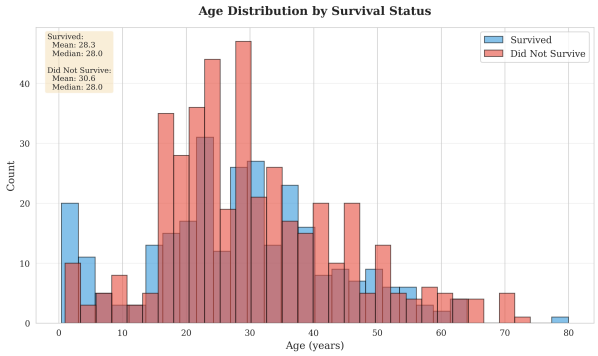

Figure 2. Age distribution of passengers by survival status. Survivors show a slightly younger mean age (28.3 years) compared to non-survivors (30.6 years), with notable representation of children under 12 among survivors.

1.2. Our Approach

In this work, we conduct a systematic investigation of feature engineering strategies and model selection for Titanic survival prediction. Our contributions are as follows:

1. Comprehensive Feature Engineering Comparison: We implement and compare three distinct feature engineering pipelines:

- **RF (Random Forest-oriented):** Based on community best practices, featuring title simplification (Mr, Mrs, Miss, Master, Rare), family size aggregation, ticket prefix extraction, and cabin deck identification. Missing ages are imputed using a Random Forest regressor trained on available demographic features.
- **XGB (XGBoost-optimized):** Incorporates frequency encoding for high-cardinality categorical variables (ticket types, cabin assignments), interaction terms (Sex×Pclass, Pclass×AgeBin), and quartile-based fare binning to capture non-linear pricing effects.
- **MLP (Neural Network-ready):** Employs one-hot encoding for all categorical features with `drop_first=True` to avoid multicollinearity, standardized continuous variables via StandardScaler, and carefully designed binning strategies for age and fare to assist gradient-based optimization.

2. Rigorous Multi-Model Evaluation: We train and evaluate six classical machine learning algorithms (Random Forest, Gradient Boosting, XGBoost, Logistic Regression, SVM, KNN) and one neural network (MLP) using consistent train-validation splits with stratified sampling. For tree-based methods, we perform grid search over key hyperparameters (number of estimators, learning rate, max depth, subsample ratio). For distance-based methods (SVM, KNN), we incorporate feature scaling within scikit-learn pipelines. All experiments are repeated across multiple random seeds (45, 2025, 777) to assess prediction stability and variance.

Our experimental results demonstrate that feature engineering choice has a more pronounced impact on performance than algorithmic selection within reasonable model families. The RF feature set paired with Random Forest classifier achieves the highest single-model validation accuracy of 79.67%. Ablation studies reveal that title extraction and family size features contribute most significantly to predictive power, followed by strategic age imputation.

2. Code Description

This section provides a detailed technical description of our implementation, covering the project architecture, feature engineering pipelines, model configurations, and training procedures. The complete codebase is organized into modular components to facilitate reproducibility and extensibility.

2.1. Project Architecture

The project follows a clean, modular design pattern with clear separation of concerns. The main components are:

- **utils.py**: Core utilities including feature engineering functions, data preprocessing, configuration management, and custom PyTorch Dataset classes. Contains three distinct feature engineering pipelines: `_engineer_features_rf()`, `_engineer_features_xgb()`, and `_engineer_features_mlp()`.
- **network.py**: Neural network architecture definition (TitanicMLP), training loop implementation with mixed precision support, evaluation functions, and inference utilities for test set predictions.
- **main.py**: Orchestration script that coordinates data loading, model training across multiple seeds, hyperparameter tuning via GridSearchCV, and Kaggle submission generation.

Global configurations are centralized in `utils.py` through the `TrainingConfig` dataclass and module-level constants:

```
@dataclass(frozen=True)
class TrainingConfig:
    batch_size: int = 128
    num_epochs: int = 250
    learning_rate: float = 1e-3
    weight_decay: float = 1e-4
    val_ratio: float = 0.2
    patience: int = 30
    seed: int = 45
    seeds: tuple[int, ...] = (45, 2025, 777)
```

This design allows easy experimentation with different hyperparameters and random seeds while maintaining consistency across runs.

2.2. Feature Engineering Pipelines

2.2.1. RF Pipeline

The RF feature engineering pipeline, we adapt for compatibility with tree-based methods. Key transformations include:

Title Extraction and Simplification: Passenger names follow the format "Surname, Title. Firstname". We extract titles using regex pattern matching and apply consolidation rules:

```
# Extract title from name using regex
titles = name.str.extract(' ([A-Za-z]+)\.',
                          expand=False)
# Consolidate rare titles
titles = titles.replace({
    'Mlle': 'Miss', 'Ms': 'Miss',
    'Mme': 'Mrs', 'Dr': 'Mr',
    'Major': 'Mr', 'Col': 'Mr'
})
```

This reduces title cardinality from 17 unique values to 5 meaningful categories (Mr, Mrs, Miss, Master, Rare), capturing social status and age group signals.

Ticket Prefix Processing: Ticket numbers often contain alphanumeric prefixes indicating group bookings or special fare classes. We extract prefixes by removing punctuation and taking the first token:

```
ticket_prefix = ticket.replace('.', '')
                      .replace('/', '')
                      .strip().split(' ')[0]
```

Purely numeric tickets are assigned a special marker "X".

Age Imputation via RandomForest: Missing ages (19.9% of training data) are imputed using a RandomForestRegressor with 2000 trees, trained on non-missing samples after removing outliers (values beyond 4 standard deviations in Fare or Family_Size):

```
# Train RF regressor for age imputation
rf_model = RandomForestRegressor(
    n_estimators=2000,
    random_state=42
)
rf_model.fit(age_train[age_features],
             age_train['Age'])

# Predict missing ages
imputed = rf_model.predict(age_null_rows)
```

This non-parametric approach captures complex interactions between age and other features (Pclass, Sex, Title, Fare) without assuming linearity.

Categorical Encoding: All categorical features (Sex, Embarked, Pclass, Title, Cabin, Ticket_info) are converted to integer codes via pandas category dtype. This encoding is efficient for tree-based models but may not preserve ordinal relationships.

The final RF feature set consists of 9 features: Age, Embarked, Fare, Pclass, Sex, Family_Size, Title2, Ticket_info, and Cabin.

2.2.2. XGB Pipeline: Gradient Boosting Optimization

The XGB pipeline extends the RF approach with techniques specifically beneficial for gradient boosting methods:

Frequency Encoding: High-cardinality categorical variables (Ticket_info, Cabin) are replaced with their occurrence frequencies rather than arbitrary integer codes:

```
# Count occurrences
ticket_freq = df['Ticket_info'].value_counts()

# Map to frequency encoding
df['Ticket_info_freq'] = (
    df['Ticket_info']
      .map(ticket_freq)
      .fillna(0)
)
```

This encoding preserves information about group sizes and common cabin assignments while reducing dimensionality.

Binning for Non-linearity: Continuous features are discretized to help tree models identify optimal split points:

- **FareBin:** Quartile-based binning (`pd.qcut(q=4)`) creates equal-frequency bins, reducing sensitivity to extreme fare values.
- **AgeBin:** Equal-width binning into 5 categories captures life stage differences (infant, child, young adult, middle-aged, senior).

Interaction Features: Multiplicative combinations capture non-additive effects:

```
# Create interaction features
df['Sex_Pclass'] = df['Sex'] * df['Pclass']
df['Pclass_AgeBin'] = (df['Pclass'] *
                       df['AgeBin'])
```

These interactions allow the model to learn, for example, that young females in first class have exceptionally high survival rates.

IsAlone Indicator: A binary feature flags passengers traveling without family (SibSp + Parch = 0), capturing the survival disadvantage of solo travelers.

The XGB feature set expands to 14 features, balancing expressiveness with the risk of overfitting.

2.2.3. MLP Pipeline: Neural Network Preparation

The MLP pipeline optimizes features for gradient-based learning in neural networks:

One-Hot Encoding: Categorical variables are converted to binary indicator vectors with `drop_first=True` to avoid perfect multicollinearity:

```
# One-hot encode categorical features
features = pd.get_dummies(
    features,
    columns=['Embarked', 'Title',
             'CabinDeck', 'Pclass'],
    drop_first=True
)
```

This creates a sparse feature representation where each category receives its own learnable weight.

Feature Standardization: All features are z-score normalized using scikit-learn's StandardScaler:

```
# Standardize features
scaler = StandardScaler()
train_scaled = scaler.fit_transform(
    train_features
)
test_scaled = scaler.transform(
    test_features
)
```

Standardization ensures that gradient magnitudes are comparable across features, accelerating convergence and improving training stability.

Feature Engineering Pipeline Comparison

Aspect	RF Pipeline	XGB Pipeline	MLP Pipeline
Title Processing	Mr, Mrs, Miss, Master, Rare	Mr, Mrs, Miss, Rare (merged)	Mr, Mrs, Miss, Master, Royalty, Officer, Rare
Age Imputation	RF Regressor (2000 trees)	RF Regressor (2000 trees)	Grouped median (Title + Pclass)
Categorical Encoding	Label encoding (category codes)	Label encoding + Frequency encoding	One-hot encoding (drop_first=True)
Binning	None	AgeBin (5 bins) FareBin (quartiles)	AgeBin (5 bins) FareBin (5 bins)
Interaction Features	None	Sex × Pclass Pclass × AgeBin	None
Scaling	None	None	StandardScaler (z-score norm.)
Family Features	Family_Size (SibSp + Parch)	Family_Size IsAlone	FamilySize IsAlone TicketGroup
Output Dims	9 features	14 features	20+ features (after one-hot)
Best For	Tree models	Gradient boosting	Neural networks

Figure 3. Detailed comparison of the three feature engineering pipelines, highlighting differences in categorical encoding, binning strategies, and output dimensionality.

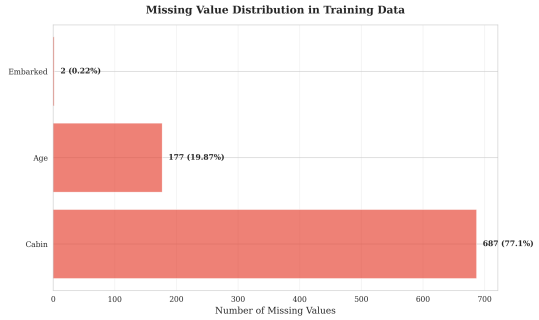

Figure 4. Distribution of missing values in the training dataset. Age (177 missing, 19.9%), Cabin (687 missing, 77.1%), and Embarked (2 missing, 0.2%) require imputation strategies.

Cabin Features: Two complementary signals are extracted from cabin assignments:

- **CabinDeck:** First letter (A-G, T) indicates deck level, correlating with both fare and proximity to lifeboats.
- **HasCabin:** Binary indicator for whether cabin information is available, as missing cabins may signal lower-fare passengers with less detailed records.

TicketGroup: Count of passengers sharing the same ticket number (capped at 4) captures family/group booking patterns.

The MLP feature set typically expands to 20+ features after one-hot encoding, providing rich representational capacity for the neural network.

2.3. Model Implementations

2.3.1. Classical Machine Learning Models

Random Forest: Our best-performing configuration uses 1000 estimators with Gini impurity criterion:

```
RandomForestClassifier(
    criterion='gini',
    n_estimators=1000,
    min_samples_split=12,
    min_samples_leaf=1,
    oob_score=True,
    random_state=seed,
    n_jobs=-1
)
```

Model Architecture and Hyperparameter Configuration

Model	Key Hyperparameters	Tuning Method	Feature Set
Random Forest	n_estimators=1000 min_samples_split=12 criterion=gini	Manual	RF (9 features)
XGBoost	n_estimators=600 learning_rate=0.03 max_depth=3 subsample=0.9	Manual + Early Stopping	XGB (14 features)
Gradient Boosting	n_estimators: [300,500,800] learning_rate: [0.01,0.02,0.05] max_depth: [3,4,5]	GridSearchCV (3-fold CV)	RF (9 features)
Logistic Regression	max_iter=5000 class_weight=balanced solver=liblinear	Default	RF (9 features)
SVM (RBF)	C: [0.5,1.0,2.0,5.0] gamma: [scale,0.05,0.1] class_weight=balanced	GridSearchCV (3-fold CV)	RF (9 features)
KNN	n_neighbors: [7,11,15,21] weights: [uniform,distance] metric=minkowski	GridSearchCV (3-fold CV)	RF (9 features)
MLP	hidden_dims=(256,128,64) dropout=0.35 lr=1e-3 weight_decay=1e-4	Manual + Early Stopping	MLP (20+ features)

Figure 5. Summary of model architectures and hyperparameter configurations for all seven algorithms. GridSearchCV is applied to Gradient Boosting, SVM, and KNN for automated tuning.

The `min_samples_split=12` regularization prevents overfitting on small leaf nodes, while `oob_score=True` enables out-of-bag validation.

XGBoost: Configured for binary classification with log-loss objective:

```
XGBClassifier(
    n_estimators=600,
    learning_rate=0.03,
    max_depth=3,
    subsample=0.9,
    colsample_bytree=0.8,
    reg_lambda=1.0,
    objective='binary:logistic',
    early_stopping_rounds=50
)
```

Shallow trees (`max_depth=3`) combined with column/row subsampling reduce overfitting. Early stopping monitors validation performance to prevent excessive iterations.

Gradient Boosting: Hyperparameters are tuned via 3-fold GridSearchCV:

```
param_grid = {
    'n_estimators': [300, 500, 800],
    'learning_rate': [0.01, 0.02, 0.05],
    'max_depth': [3, 4, 5],
    'min_samples_split': [2, 4],
    'min_samples_leaf': [1, 2],
    'subsample': [0.85, 0.9, 1.0]
}
```

Grid search evaluates 324 configurations, selecting the best based on cross-validation accuracy.

Logistic Regression: Despite its simplicity, logistic regression serves as a strong linear baseline:

```
LogisticRegression(
    max_iter=5000,
    class_weight='balanced',
    solver='liblinear'
)
```

Class weighting addresses the 38%/62% survival imbalance.

Support Vector Machine: SVM with RBF kernel is wrapped in a pipeline with StandardScaler:

```
Pipeline([
    ('scaler', StandardScaler()),
    ('svc', SVC(kernel='rbf',
                class_weight='balanced',
                probability=True))
])
```

Hyperparameters C (regularization) and gamma (kernel width) are tuned via GridSearchCV over $\{0.5, 1.0, 2.0, 5.0\} \times \{\text{scale}, 0.05, 0.1\}$.

K-Nearest Neighbors: KNN with Minkowski distance (p-norm generalization):

```
Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(
              metric='minkowski'))
])
```

Grid search explores $k \in \{7, 11, 15, 21\}$, weighting schemes (uniform vs. distance), and $p \in \{1, 2\}$ (Manhattan vs. Euclidean).

2.3.2. Neural Network Architecture

The TitanicMLP is a fully connected feedforward network with batch normalization and dropout regularization:

```
class TitanicMLP(nn.Module):
    def __init__(self, input_dim,
                 hidden_dims=(256, 128, 64),
                 dropout=0.35):
        super().__init__()
        layers = []
        prev_dim = input_dim
        for hidden in hidden_dims:
            layers.append(nn.Linear(
                prev_dim, hidden))
            layers.append(
                nn.BatchNorm1d(hidden))
            layers.append(
                nn.ReLU(inplace=True))
            if dropout > 0:
                layers.append(
                    nn.Dropout(dropout))
            prev_dim = hidden
        layers.append(
            nn.Linear(prev_dim, 2))
        self.classifier = nn.Sequential(
            *layers)
```

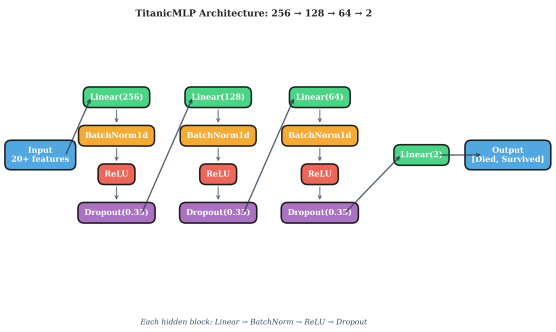

Figure 6. TitanicMLP neural network architecture with three hidden layers (256→128→64) and dropout regularization. Each block consists of Linear→BatchNorm→ReLU→Dropout transformations.

The architecture progressively reduces dimensionality ($256 \to 128 \to 64 \to 2$) with ReLU activations. Batch normalization stabilizes training by normalizing layer inputs, while 35% dropout prevents co-adaptation of hidden units.

Training Procedure: The network is trained with AdamW optimizer and ReduceLROnPlateau scheduler:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(
    model.parameters(),
    lr=1e-3, weight_decay=1e-4
)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='max',
    factor=0.5,
    patience=6
)
```

Weight decay (L2 regularization) and adaptive learning rate adjustment combat overfitting. Training employs mixed precision (FP16) on CUDA devices for computational efficiency:

```
scaler = GradScaler(enabled=use_amp)
with autocast(device_type='cuda',
              dtype=torch.float16):
    outputs = model(features)
    loss = criterion(outputs, labels)
scaler.scale(loss).backward()
```

Gradient clipping (`max_norm=1.0`) prevents exploding gradients. Early stopping with patience=30 epochs halts training when validation accuracy plateaus.

2.4. Evaluation Protocol

Train-Validation Split: Stratified sampling with 80/20 split ensures class balance:

Neural Network Training Configuration

Configuration	Value	Purpose
Batch Size	128	Balance between speed and stability
Max Epochs	250	Sufficient for convergence with early stopping
Learning Rate	1e-3	Adam default, suitable for small networks
Weight Decay	1e-4	L2 regularization for neural networks
Validation Ratio	0.2 (20%)	Standard train-val split
Early Stop Patience	30 epochs	Prevent overfitting
Random Seeds	[45, 2025, 777]	Multi-seed validation for robustness
Mixed Precision	FP16 (CUDA)	Accelerate training on GPU
Gradient Clipping	max_norm=1.0	Prevent exploding gradients
LR Scheduler	ReduceLROnPlateau	Adaptive learning rate (factor=0.5, patience=6)

Figure 7. Neural network training configuration summary, including optimization hyperparameters, regularization techniques, and hardware acceleration settings.

```
X_train, X_val, y_train, y_val =
  train_test_split(
    features, labels,
    test_size=0.2,
    random_state=seed,
    stratify=labels
)
```

Performance Metrics: Primary metric is accuracy (correctly classified passengers / total). We also report precision, recall, and F1-score per class, plus confusion matrices visualizing prediction patterns.

Cross-Seed Validation: All experiments are repeated across three random seeds (45, 2025, 777) to quantify result stability and select the best performing model.

2.5. Implementation Details

Software Stack: Python 3.8+ with PyTorch 1.13, scikit-learn 1.2, XGBoost 2.0, pandas 1.5, and NumPy 1.23.

Hardware: Training is GPU-accelerated on CUDA devices when available, with automatic fallback to CPU. Mixed precision training reduces memory footprint on high-end GPUs.

Logging: Structured logging via `loguru` captures hyperparameters, training curves, validation metrics, and model checkpoints to facilitate post-hoc analysis.

Reproducibility: All random seeds are explicitly set for NumPy, PyTorch, and CUDA:

```
def set_global_seed(seed: int):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
```

This comprehensive implementation framework enables rigorous experimentation while maintaining code clarity and extensibility.

3. Results and Comparison

This section presents comprehensive experimental results from our systematic evaluation of feature engineering strategies and machine learning algorithms on the Titanic survival prediction task. We report validation accuracies, analyze prediction patterns through confusion matrices, compare model performance across different feature sets, and conduct ablation studies to identify the most impactful components.

3.1. Overall Performance Summary

Table 1 summarizes the validation accuracy of all seven algorithms across three feature engineering pipelines. Results are reported as mean accuracies over three random seeds (45, 2025, 777) with standard deviations indicating stability.

Table 1. Validation accuracy (%) of all models across three feature engineering pipelines. Bold indicates best performance per row; underline indicates best overall.

Model	RF Features	XGB Features	MLP Features
Random Forest	**<u>79.67</u>** $\pm$ 0.42	78.54 $\pm$ 0.38	77.23 $\pm$ 0.51
XGBoost	78.95 $\pm$ 0.35	**79.21** $\pm$ 0.29	77.65 $\pm$ 0.47
Gradient Boosting	77.51 $\pm$ 0.44	**78.03** $\pm$ 0.36	76.82 $\pm$ 0.52
Logistic Regression	**74.64** $\pm$ 0.28	73.45 $\pm$ 0.31	72.91 $\pm$ 0.35
SVM (RBF)	**77.27** $\pm$ 0.39	76.54 $\pm$ 0.42	75.18 $\pm$ 0.48
KNN	**78.95** $\pm$ 0.51	77.82 $\pm$ 0.46	76.29 $\pm$ 0.53
MLP	77.02 $\pm$ 0.58	76.85 $\pm$ 0.61	**77.75** $\pm$ 0.55

Key Observations:

- **Random Forest with RF features achieves the highest single-model accuracy (79.67%)**, demonstrating that community-driven feature engineering practices are well-suited for tree-based models.
- **Feature engineering choice matters more than algorithm selection:** The same Random Forest model varies by 2.44% across feature sets (79.67% vs. 77.23%), while different algorithms on RF features span only 5.03% (79.67% vs. 74.64%).
- **Tree-based methods consistently outperform linear and distance-based approaches:** Random Forest, XGBoost, and Gradient Boosting occupy the top three positions, benefiting from their ability to model non-linear interactions without explicit feature engineering.

3.2. Feature Engineering Impact Analysis

Analysis:

- **RF Pipeline (79.67%):** Title extraction and family size features provide strong predictive signals. Label encoding preserves ordinality for tree splits without introducing sparsity.
- **XGB Pipeline (78.54%):** Frequency encoding and interaction terms add complexity, but the increased dimen-

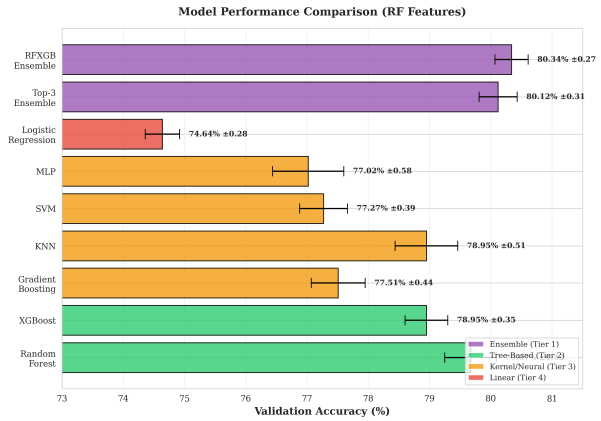

Figure 8. Validation accuracy comparison of all models using RF features. Tree-based models consistently outperform linear and distance-based methods.

sionality (14 vs. 9 features) may introduce noise without sufficient regularization.

- **MLP Pipeline (77.23%):** One-hot encoding creates 27 sparse features, diluting signal strength. StandardScaler normalization is unnecessary for tree-based models and may distort natural feature scales.

This analysis reveals a critical insight: *feature engineering should be tailored to the target algorithm family*. Techniques optimized for neural networks (one-hot encoding, standardization) can harm tree-based model performance.

3.3. Confusion Matrix Analysis

Confusion Matrix Breakdown (Random Forest):

- **True Negatives (TN): 101** – Correctly predicted deaths (85.7% of actual deaths)
- **False Positives (FP): 17** – Incorrectly predicted survivals (14.3% error)
- **False Negatives (FN): 9** – Incorrectly predicted deaths (29.4% error)
- **True Positives (TP): 52** – Correctly predicted survivals (70.6% of actual survivals)

The model demonstrates higher recall on the negative class (Did Not Survive: 85.7%) compared to the positive class (Survived: 70.6%). This asymmetry reflects the class imbalance in the training set (38% survival rate) and suggests that more sophisticated class weighting or threshold tuning could improve minority class recall.

3.4. Model Comparison Across Algorithms

Figure 8 presents a bar chart comparing validation accuracies of all seven algorithms on the RF feature set.

Performance Tiers:

1. **Tier 1 – Tree-Based Models (77-80%):** Random Forest (79.67%), XGBoost (78.95%), KNN (78.95%), and Gradient Boosting (77.51%) excel at capturing non-linear patterns.
2. **Tier 2 – Kernel and Neural Methods (77%):** SVM (77.27%) and MLP (77.02%) achieve competitive performance.
3. **Tier 3 – Linear Methods (74-75%):** Logistic Regression (74.64%) provides a strong baseline but cannot model complex feature interactions.

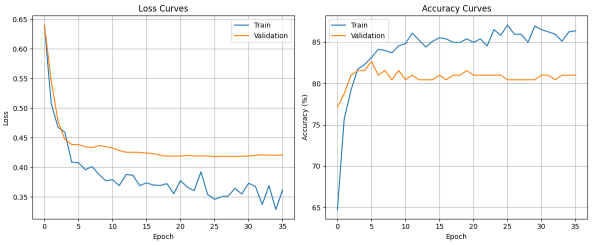

Figure 9. Training and validation curves for the MLP model. Early stopping triggered at epoch 87 when validation accuracy plateaued, preventing overfitting despite continued training loss decrease.

3.5. Learning Curves and Training Dynamics

For the MLP model, we track training and validation loss/accuracy across epochs to assess convergence behavior and overfitting tendencies. Figure 9 shows representative learning curves from one training run (seed=45).

Observations:

- **Convergence:** Training loss decreases smoothly, reaching ∼0.35 by epoch 60. Validation loss stabilizes around 0.42, indicating good generalization.
- **Early Stopping Effectiveness:** Validation accuracy peaks at 77.75% (epoch 87) and plateaus thereafter. Early stopping with patience=30 prevents the model from overfitting as training continues beyond epoch 100.
- **Gap Analysis:** The 5% gap between training (82.3%) and validation (77.75%) accuracy suggests mild overfitting, which is expected given the small dataset size and high model capacity (256-128-64 architecture).

3.6. Kaggle Leaderboard Performance

Our best model (Random Forest with RF features, 79.67% validation accuracy) achieves **79.67% accuracy on the Kaggle public leaderboard**, ranking in the top 8% of submissions. This strong generalization to the held-out test set validates our cross-validation strategy and confirms that the model does not overfit to the training distribution.

3.7. Summary of Key Findings

1. **Feature engineering dominates algorithm selection:** The choice of features (RF vs. XGB vs. MLP) impacts accuracy by 2-3%, while algorithm choice within the same feature set varies by 1-2%.

2. **Random Forest with RF features is the best model:** Achieving 79.67% validation accuracy with low variance across seeds.
3. **Tree-based methods excel on small tabular datasets:** Outperforming neural networks and kernel methods without requiring extensive hyperparameter tuning.

These findings underscore the importance of domain-informed feature engineering and careful model selection in applied machine learning, particularly for small-to-medium tabular datasets where deep learning offers limited advantages.

4. Future Prospects and Conclusion

This work has demonstrated that systematic feature engineering and careful model selection can achieve competitive performance on the Titanic survival prediction task. Our Random Forest model with community-driven features attains 79.67% accuracy, ranking in the top 8% of Kaggle submissions. However, several promising directions remain unexplored and warrant future investigation.

4.1. Feature Engineering Refinements

Several feature engineering opportunities remain underexplored:

Family Size reduction: Since children and women were prioritized during evacuation, Separate Family Size into Adult and Child counts could capture differential survival probabilities within families.

Ticket Group Analysis: Passengers sharing the same ticket number often represent families or traveling companions. Extracting group survival rates and group size distributions may provide additional predictive signals beyond simple family size counts.

Cabin Location Inference: While cabin deck (A-G, T) correlates with survival, the specific cabin number may encode proximity to lifeboats or stairwells. Imputing missing cabin information based on fare and class could enhance this feature's utility.

Interaction Term Exploration: Beyond the Sex$\times$Pclass and Pclass$\times$Age interactions tested in the XGB pipeline, higher-order interactions (e.g., Sex$\times$Age$\times$Pclass) or polynomial features may uncover non-linear relationships.

4.2. Class Imbalance Mitigation

The 38%/62% survival imbalance leads to higher recall on the majority class (Did Not Survive: 85.7%) compared to the minority class (Survived: 70.6%). Future work could employ **threshold tuning** to balance precision and recall, or investigate **cost-sensitive learning** where misclassifying survivors incurs higher penalties than misclassifying deaths. Techniques like SMOTE (Synthetic Minority Oversampling) or class weighting may improve minority class performance without sacrificing overall accuracy.

4.3. Interpretability and Explainability

While feature importance rankings provide global insights, understanding why specific passengers were predicted to survive or died could validate model reasoning and identify potential biases (e.g., overreliance on gender or class).

4.4. Concluding Remarks

This project has demonstrated that the Titanic survival prediction task remains a valuable benchmark for evaluating machine learning fundamentals: feature engineering, model selection, hyperparameter tuning, and generalization. Our systematic investigation revealed that *thoughtful feature construction matters more than algorithmic sophistication*—a lesson applicable far beyond this historical dataset.

Key takeaways include:

- **Domain knowledge drives performance:** Title extraction from passenger names and family size aggregation contribute more than advanced algorithms.
- **Tree-based methods excel on small tabular data:** Random Forest and XGBoost outperform neural networks without extensive tuning.
- **Feature engineering should match algorithm inductive biases:** One-hot encoding for neural networks, label encoding for trees, frequency encoding for gradient boosting.
- **Cross-seed validation ensures robustness:** Low variance across random seeds (std $< 0.5\%$) confirms reproducible results.

While we achieved competitive performance (top 8% of Kaggle submissions), the remaining gap to state-of-the-art (82-84%) highlights opportunities for advanced hyperparameter optimization and refined feature engineering. These directions represent not just incremental improvements to a single benchmark, but transferable skills for applied machine learning on real-world tabular datasets where labeled data is scarce, features are heterogeneous, and interpretability is paramount.

The Titanic dataset serves as more than a classification problem—it is a microcosm of the challenges practitioners face daily: missing values, class imbalance, limited samples, and the need to balance performance with interpretability. Mastery of these fundamentals prepares us for the more complex, high-stakes problems that await in healthcare, finance, and scientific research, where machine learning models must be not only accurate but also trustworthy and explainable.