Towards AI

Member-only story

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient Boosting Internal Optimization for Superior Predictive Performance

Shenggang Li · Following

Published in Towards AI · 15 min read · Apr 1, 2025

Photo by Austin Neill on Unsplash

Introduction

In this post, I demonstrate how reinforcement learning (*RL*) can directly enhance the performance of gradient boosting models (*GBM*) by dynamically adjusting the learning rate at each boosting iteration. Unlike traditional approaches — such as hyperparameter tuning, model blending, or architecture searches — my method integrates *RL* directly into the gradient boosting procedure. Specifically, the *RL* agent actively determines an optimal learning rate to scale the gradient updates of each new tree, adaptively controlling its contribution to the final model. This dynamic adjustment enables *GBM* to efficiently respond to evolving data patterns, significantly boosting performance.

To validate this approach, I conducted experiments using both synthetic and *Kaggle* benchmark datasets. For regression tasks, the *RL*-guided GBM consistently outperformed competitive models like *XGBoost, LightGBM*, and Random Forest. For classification tasks, the method achieved accuracy comparable to state-of-the-art models such as *LightGBM* and *XGBoost*, while clearly outperforming *Random Forest* and *AdaBoost*. I include detailed Python code demonstrating precisely how the *RL* agent selects optimal gradient scaling factors dynamically during training, ensuring each boosting iteration is strategically tailored to maximize predictive accuracy.

There is considerable potential to further refine this method. Future improvements could involve adopting more sophisticated *RL* algorithms or designing specialized tree architectures optimized for *RL*-driven boosting. Ultimately, incorporating *RL* directly at the gradient-scaling stage introduces a novel pathway for boosting algorithms, enabling smarter, more adaptive modeling strategies and unlocking new possibilities for achieving superior model performance.

Mechanism of RL-Enhanced Gradient Boosting for Regression

Overview of Gradient Boosting

Gradient boosting is a powerful ensemble method that sequentially combines weak learners — typically decision trees — to form a strong predictive model. At each iteration t, a new model *h_t(x)* is fitted to the residuals (or “pseudo-residuals”) of the current ensemble. Mathematically, given training data:

$$\{(x_i, y_i)\}_{i=1}^{N}$$

and an initial model *F_0(x)* (often a constant), the algorithm updates the model as follows:

$$F_t(x) = F_{t-1}(x) + \alpha_t h_t(x)$$

where *α_t* is a learning rate and *h_t(x)* is the new weak learner fitted to the residuals:

$$r_{i,t} = y_i - F_{t-1}(x_i)$$

This process minimizes a loss function *L(y, F(x))*, typically via gradient descent. The weight *α_t* can be computed by solving an optimization problem (e.g., using line search) to minimize the loss with respect to the addition of *h_t(x)*.

Mechanism of Boosting

In traditional gradient boosting, the weak learner *h_t(x)* is chosen to approximate the negative gradient of the loss function:

$$-\nabla_F L(y_i, F(x_i))$$

For regression tasks with squared error loss, the pseudo-residuals are simply:

$$r_{i,t} = y_i - F_{t-1}(x_i)$$

and the weak learner is trained to predict *r_{i, t}*. The final prediction is the sum of all learners' contributions:

$$\hat{y}(x) = F_0(x) + \sum_{t=1}^{M} \alpha_t h_t(x)$$

In classification settings, a logistic loss or exponential loss is commonly used, and the boosting algorithm adapts accordingly. The strength of gradient boosting lies in its ability to focus subsequent learners on instances that previous learners mispredicted, thereby iteratively reducing the overall error.

Integrating Reinforcement Learning into Boosting

The novel idea behind *RL*-enhanced boosting is to integrate reinforcement learning (*RL*) into the boosting process. Rather than using a fixed learning rate α_t or computing it via a deterministic line search, we allow an *RL* agent to dynamically choose a multiplier for *α_t* at each iteration.
Mathematically, suppose the standard learning rate computed by boosting is *α_t*. The RL agent then chooses a multiplier mmm from a discrete set (e.g., *{0.8,0.9,1.0,1.1,1.2}*). The effective learning rate becomes:

$$\alpha_t = m \cdot \alpha_t^{\text{std}}$$

The *RL* agent's objective is to learn a policy that selects mmm such that the overall predictive performance improves. The state used by the *RL* agent is defined by a discretization of the current weighted training error. For example, if the error is *err_t*, then the state might be:

$$s_t = \min\left(\lfloor err_t \times 10 \rfloor, 9\right)$$

This simple mapping transforms a continuous error into one of 10 discrete states. The reward is computed as the change in validation performance from one iteration to the next. In our current formulation (prior versions), the reward was defined as the decrease in validation error. However, to improve performance we can also target *AUC* or *KS*; in this version we focus on minimizing error, so we define:

$$r_t = 10 \times (E_{t-1} - E_t)$$

where *E_t* is the validation error at iteration *t*. A positive reward indicates a reduction in error.

RL Algorithm and Q-Learning Integration

Once we define the state *s_t*, action *a_t* (multiplier choice), and reward *r_t*, we can use *Q-learning* to update a *Q-table*. The *Q-learning* update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_q \left[r_t + \gamma_q \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

where α_q is the learning rate for the *Q-table* and γ is the discount factor. The algorithm uses an epsilon-greedy strategy to balance exploration and exploitation. In each boosting iteration, the *RL* agent samples an action a_t either randomly (with probability ϵ) or by selecting the action with the highest *Q-value* for the current state. This multiplier m_t is then applied to scale the standard boosting learning rate. The final ensemble prediction is updated as:

$$F_t(x) = F_{t-1}(x) + m_t \cdot \alpha_t^{\text{std}} \cdot h_t(x)$$

This process is repeated for M iterations. The resulting *Q-table* should capture which multiplier is most beneficial in different states, dynamically adjusting the contribution of each weak learner based on the current training error.

Advantages, Improvements, and Practical Considerations

Incorporating *RL* into gradient boosting offers several advantages. First, the dynamic selection of the learning rate multiplier allows the model to adapt to different learning stages. For example, when the training error is high, a lower multiplier might help avoid overfitting; conversely, when the model begins to converge, a higher multiplier may accelerate improvements. This dynamic adjustment can lead to better convergence properties and overall performance.

Moreover, using *Q-learning* to guide multiplier selection adds an element of self-tuning that may improve robustness across different datasets. The *RL* agent learns from the validation performance, effectively optimizing the boosting process in a data-driven manner. This is particularly useful when

the optimal learning rate varies over time or across regions of the feature space.

However, the approach has its challenges. One key aspect is the discretization of the error into states. A coarse discretization might lose subtle nuances, while an overly fine discretization may lead to sparse observations for each state, impeding learning. Future improvements could involve adaptive state discretization or even using continuous state representations with function approximation methods.

Another area for improvement is the reward function. Although we defined the reward based on the decrease in validation error (or another metric like *AUC* or *KS*), further refinement could involve multi-objective rewards that consider both error reduction and stability, or even incorporating risk measures into the reward signal. Advanced techniques such as Proximal Policy Optimization (*PPO*) or actor-critic methods could further stabilize and enhance learning compared to the simple *Q-learning* approach.

From a practical standpoint, integrating *RL* into boosting does add computational overhead. It is essential to ensure that the *RL* component converges reliably. Techniques such as experience replay, target networks, or hyperparameter tuning (e.g., adjusting ϵ, α_q, and γ_q) can help mitigate these issues. Moreover, cross-validation and walk-forward testing are recommended to verify that the *RL*-enhanced boosting consistently outperforms standard methods across different market conditions or datasets.

Experiment: RL-Enhanced GBM Regression

Case Study 1: Predicting Used Car Prices from Kaggle Dataset

In this experiment, I applied my proposed Reinforcement Learning (*RL*)-boosted Gradient Boosting Model (*GBM*) regression to predict used car prices using a *Kaggle* Playground Series dataset (link). The goal was to accurately estimate the continuous target — car price — from features like mileage, brand, transmission, engine specs, and accident history, making it perfect for testing the *RL-GBM* regression approach.

First, I cleaned the data by handling missing values, extracting numeric features from text (e.g., horsepower and engine size), and encoding categorical variables with one-hot encoding. Then, I identified the top 20 features most correlated with price, streamlining the dataset to ensure the model focuses only on the strongest predictors.

Lastly, I compared the *RL*-enhanced *GBM* regression model against powerful benchmarks such as *XGBoost, LightGBM*, and *Random Forest*. Initial tests showed promising results, highlighting that integrating *RL* into *GBM* significantly boosts predictive accuracy, opening opportunities for further improvement.

```
import re
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
import lightgbm as lgb
import matplotlib.pyplot as plt

# === Load Data: car price: https://www.kaggle.com/competitions/playground-serie
df = pd.read_csv("train.csv")
```

```
# === Initial Cleanup ===
# Strip whitespace from column names
df.columns = df.columns.str.strip()

# Drop rows where target (price) is missing or 0
df = df[df['price'].notnull() & (df['price'] > 0)]

# === Clean and Extract Numeric Info ===
def extract_hp(text):
    match = re.search(r"(\d+\.?\d*)\s*HP", str(text))
    return float(match.group(1)) if match else np.nan

def extract_engine_size(text):
    match = re.search(r"(\d+\.\d+)L", str(text))
    return float(match.group(1)) if match else np.nan

def extract_cylinder_count(text):
    match = re.search(r"(\d+)\s*[Vv]?\s*[Cc]ylinder", str(text))
    return int(match.group(1)) if match else np.nan

df['engine_hp'] = df['engine'].apply(extract_hp)
df['engine_L'] = df['engine'].apply(extract_engine_size)
df['cylinder'] = df['engine'].apply(extract_cylinder_count)

# Drop original engine column
df.drop(columns=['engine'], inplace=True)

# === Step 4: Handle Missing Values ===
# Add missing flags for selected columns
for col in ['int_col', 'transmission']:
    df[f'flag_{col}_missing'] = df[col].isnull().astype(int)

# Fill missing with placeholder
df['int_col'] = df['int_col'].fillna('Missing')
df['transmission'] = df['transmission'].fillna('Missing')

# Fill numeric engine values
for col in ['engine_hp', 'engine_L', 'cylinder']:
    df[col] = df[col].fillna(df[col].median())

# === Step 5: Categorical One-hot Encoding ===
cat_cols = ['brand', 'model', 'fuel_type', 'transmission', 'ext_col', 'int_col',
df = pd.get_dummies(df, columns=cat_cols, drop_first=True)

# === Drop unnecessary or ID fields ===
df.drop(columns=['id'], inplace=True)

# === Correlation with Target (price) ===
target = 'price'
features = [col for col in df.columns if col != target]
```

```
corr_values = df[features].apply(lambda x: x.corr(df[target]))
abs_corr = corr_values.abs().sort_values(ascending=False)

# Select top N features
top_n = 20
top_features = abs_corr.head(top_n).index.tolist()

# === Final Model Data ===
model_df = df[top_features + [target]].dropna()

features = ['milage',  'model_year',  'engine_hp',
 'accident_None reported',  'brand_Lamborghini',
 'transmission_A/T',  'engine_L',  'cylinder',
 'brand_Bentley',  'brand_Porsche',
 'int_col_Nero Ade',  'transmission_7-Speed Automatic with Auto-Shift',
 'transmission_6-Speed A/T',
 'transmission_8-Speed Automatic with Auto-Shift',
 'int_col_Gray', 'int_col_Beige',  'transmission_8-Speed Automatic',
 'model_911 GT3',  'brand_Rolls-Royce',
 'transmission_8-Speed A/T']

model_df['price'] = np.sqrt(model_df['price'] + 1)

print("Selected Features:\n", top_features)
print("\nCleaned Model DataFrame shape:", model_df.shape)
```

The following algorithm combines reinforcement learning (*RL*) with gradient boosting methods (*GBM*), dynamically adjusting the learning rate for each new decision tree to optimize predictive performance.

The key steps include:

Initialization:

- Set initial predictions to zero and define a discrete action space representing possible multipliers for the base learning rate.
- Initialize a *Q-table* to guide action (learning rate multiplier) selection based on model state.

Gradient Boosting with RL-driven Learning Rate:

variance.

- Use an *RL* agent (*Q-learning*) to select an optimal learning rate multiplier, dynamically scaling the contribution of each tree.

State Definition for *RL*:

- Represent the state by discretizing the average magnitude of residual errors, allowing *RL* to react adaptively based on prediction difficulty.

Reward Calculation and Q-learning Update:

- Compute the reward as improvement in validation prediction accuracy (reduction in validation *MAPE*).
- Update the *Q-table* using standard *Q-learning* formulas to reinforce effective learning rate decisions.

Feature Importance Aggregation and Model Evaluation:

- Aggregate feature importance across iterations, scaled by dynamically selected learning rates.
- Continuously track and report performance metrics (*MAPE*) on training, validation, and testing datasets.

Through these steps, the *RL-GBM* method will optimize gradient updates, improving prediction accuracy and adapting efficiently to data complexity.

```
#----------------GBM regrssion---------------
##############################################
# Metric Functions
##############################################
def mean_absolute_percentage_error(y_true, y_pred):
    eps = 1e-6
    return np.mean(np.abs(y_true - y_pred) / (np.abs(y_true) + eps)) * 100.0

def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

##############################################
# RL-GBM Regressor
##############################################
def rl_gbm_regressor(X_train, y_train, X_val, y_val, X_test, y_test,
                     base_learn_rate=0.1, M=200,
                     max_depth=3,
                     actions=[0.4, 0.6, 0.8, 0.9, 1.0, 1.1, 1.2]):
    n_train = len(X_train)
    n_features = X_train.shape[1]
    feature_names = X_train.columns

    num_states = 10
    num_actions = len(actions)
    Q = np.zeros((num_states, num_actions))
    epsilon = 0.1
    alpha_q = 0.1
    gamma_q = 0.9

    F_train = np.zeros(n_train)
    F_val   = np.zeros(len(X_val))
    F_test  = np.zeros(len(X_test))
    feature_importance_sum = np.zeros(n_features, dtype=np.float64)

    def get_state(g):
        avg_g = np.mean(np.abs(g))
        idx = int(avg_g / 10.0)
        return min(idx, num_states - 1)

    prev_val_mape = mean_absolute_percentage_error(y_val, F_val)
    train_mape_hist, val_mape_hist, test_mape_hist = [], [], []
```

```
    for t in range(M):
        g_train = y_train - F_train
        tree = DecisionTreeRegressor(max_depth=max_depth, random_state=1)
        tree.fit(X_train, g_train)
        tree_importance = tree.feature_importances_

        g_pred_train = tree.predict(X_train)
        g_pred_val   = tree.predict(X_val)
        g_pred_test  = tree.predict(X_test)

        state = get_state(g_train)
        action_idx = np.random.randint(num_actions) if np.random.rand() < epsilo
        alpha_eff = base_learn_rate * actions[action_idx]

        feature_importance_sum += alpha_eff * tree_importance

        F_train += alpha_eff * g_pred_train
        F_val   += alpha_eff * g_pred_val
        F_test  += alpha_eff * g_pred_test

        val_mape_now = mean_absolute_percentage_error(y_val, F_val)
        reward = prev_val_mape - val_mape_now
        prev_val_mape = val_mape_now

        next_state = get_state(y_train - F_train)
        Q[state, action_idx] += alpha_q * (reward + gamma_q * np.max(Q[next_stat

        train_mape_hist.append(mean_absolute_percentage_error(y_train, F_train))
        val_mape_hist.append(val_mape_now)
        test_mape_hist.append(mean_absolute_percentage_error(y_test, F_test))

    total = feature_importance_sum.sum()
    if total > 0:
        feature_importance_sum /= total

    feat_importance_df = pd.DataFrame({
        'Feature': feature_names,
        'RLGBM_Importance': feature_importance_sum
    }).sort_values('RLGBM_Importance', ascending=False).reset_index(drop=True)

    print("\n=== RL-GBM Feature Importances ===")
    print(feat_importance_df)

    return F_test, Q, train_mape_hist, val_mape_hist, test_mape_hist

##############################################
# Load and Prepare Real Car Data
##############################################
```

```
# Feature columns and target

car_df = model_df.copy()
feature_cols = features[:]
target_col = 'price'

# Drop rows with missing values
car_df = car_df.dropna(subset=feature_cols + [target_col])

# Split data
X = car_df[feature_cols]
y = car_df[target_col]
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2,
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test

##############################################
# Run RL-GBM
##############################################
final_test_preds, Q_table, train_mape_hist, val_mape_hist, test_mape_hist = rl_g
    X_train, y_train, X_val, y_val, X_test, y_test,
    base_learn_rate=0.03, M=300, max_depth=4
)


final_test_preds, Q_table, train_mape_hist, val_mape_hist, test_mape_hist = rl_g
    X_train, y_train, X_val, y_val, X_test, y_test,
    base_learn_rate=0.02, M=400, max_depth=5
)

# Metrics
mape_rl  = mean_absolute_percentage_error(y_test, final_test_preds)
rmse_rl  = rmse(y_test, final_test_preds)
print("\n=== RL-GBM Regressor Results ===")
print(f"MAPE(%): {mape_rl:.4f}, RMSE: {rmse_rl:.4f}")

##############################################
# Compare with XGB, LGB, RF
##############################################
xgb_reg = xgb.XGBRegressor(n_estimators=150, max_depth=3, learning_rate=0.1, obj
xgb_reg.fit(X_train, y_train)
xgb_preds = xgb_reg.predict(X_test)
xgb_mape  = mean_absolute_percentage_error(y_test, xgb_preds)
xgb_rmse_ = rmse(y_test, xgb_preds)

lgb_reg = lgb.LGBMRegressor(n_estimators=150, max_depth=3, learning_rate=0.1, ra
lgb_reg.fit(X_train, y_train)
lgb_preds = lgb_reg.predict(X_test)
lgb_mape  = mean_absolute_percentage_error(y_test, lgb_preds)
lgb_rmse_ = rmse(y_test, lgb_preds)
```

```
rf_reg = RandomForestRegressor(n_estimators=150, max_depth=10, random_state=42)
rf_reg.fit(X_train, y_train)
rf_preds = rf_reg.predict(X_test)
rf_mape  = mean_absolute_percentage_error(y_test, rf_preds)
rf_rmse_ = rmse(y_test, rf_preds)

print("\n=== Performance Comparison ===")
print(f"RL-GBM => MAPE: {mape_rl:.4f}  RMSE: {rmse_rl:.4f}")
print(f"XGBoost => MAPE: {xgb_mape:.4f}  RMSE: {xgb_rmse_:.4f}")
print(f"LightGBM => MAPE: {lgb_mape:.4f}  RMSE: {lgb_rmse_:.4f}")
print(f"RandomForest => MAPE: {rf_mape:.4f}  RMSE: {rf_rmse_:.4f}")

# Plot MAPE over iterations
plt.figure(figsize=(8, 6))
plt.plot(train_mape_hist, label="Train MAPE", linewidth=2)
plt.plot(val_mape_hist, label="Val MAPE", linewidth=2)
plt.plot(test_mape_hist, label="Test MAPE", linewidth=2)
plt.xlabel("Boosting Iteration")
plt.ylabel("MAPE (%)")
plt.title("RL-GBM Regressor MAPE Over Iterations")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

The result below summarizes the performance of *RL-GBM* compared to *XGBoost, LightGBM*, and *Random Forest*:

```
#########RL GBM regrssion#########################
=== Performance Comparison ===
RL-GBM => MAPE: 18.9071  RMSE: 66.1632
XGBoost => MAPE: 19.3608  RMSE: 66.3405
LightGBM => MAPE: 19.3419  RMSE: 66.2819
RandomForest => MAPE: 19.2944  RMSE: 66.8720
```

The results show that the proposed *RL-GBM* method outperforms traditional gradient boosting models and Random Forest, achieving the lowest mean

absolute percentage error (*MAPE*) and root mean square error (*RMSE*). This indicates the effectiveness of integrating reinforcement learning for dynamically optimizing the learning rate in gradient boosting frameworks.

Case Study 2: Predicting Customer Purchase Decisions

To further validate the effectiveness of the *RL*-enhanced *GBM* regression method, I conducted another experiment using a synthetic marketing campaign dataset available at https://github.com/datalev001/RL_GBM. To ensure privacy, the data used here is synthetic but realistic, featuring important customer attributes like age, income, recent purchase days, loyalty scores, holiday indicators, and preferred shopping channels.

Here are the results:

```
=== RL-GBM Feature Importances ===
          Feature  RLGBM_Importance
0          Income          0.485857
1             Age          0.294284
2            Days          0.139355
3         Holiday          0.079364
4         Loyalty          0.001118
5  Channel_Mobile          0.000021
6  Channel_Online          0.000000

=== RL-GBM Regressor Results ===
MAPE(%): 1.7919, RMSE: 103.4443

=== Performance Comparison ===
RL-GBM => MAPE: 1.7919  RMSE: 103.4443
XGBoost => MAPE: 2.1236  RMSE: 147.1589
LightGBM => MAPE: 2.0508  RMSE: 143.5815
RandomForest => MAPE: 2.1796  RMSE: 154.4802
```

Analyzing feature importance revealed that income (*48.6%*) and age (*29.4%*) were the strongest predictors of purchase decisions. Days since last purchase (*13.9%*) and holiday indicators (*7.9%*) also contributed notably, while loyalty scores and shopping channels had minimal influence.

When comparing model performance, the *RL-GBM* regression clearly outperformed traditional methods. It achieved the lowest error rates — *MAPE* of *1.79%* and *RMSE* of *103.44* — while other models like *XGBoost*, *LightGBM*, and RandomForest lagged behind, showing significantly higher errors (*MAPE* ranging from *2.05%* to *2.18%*, *RMSE* between *143.58* and *154.48*). These results confirm the strong predictive capability and reliability of the *RL*-enhanced *GBM* regression approach.

Experiment: RL-Enhanced GBM Classifiers

In this experiment, I'm extending the *RL*-boosted Gradient Boosting Model (*GBM*) from regression to classification tasks. While the *RL-GBM* regression method worked well on continuous outcomes like predicting car prices and customer spending, this section focuses on binary targets — things like whether a car's price is above average or if a customer will buy something.

The key difference here is that the *RL* strategy guides *GBM* to pick splits specifically optimized for classification metrics, such as *AUC* or *KS*, rather than just reducing regression errors. This targeted approach helps the model find sharper decision boundaries, leading to better predictions.

I'll test this *RL*-enhanced *GBM* classifier on the same *Kaggle* car price dataset and the synthetic marketing data, comparing it against popular classifiers

like *XGBoost, LightGBM*, *AdaBoost*. . The complete code and datasets are accessible at:
https://github.com/datalev001/RL_GBM

Predicting Customer Purchase Decisions Using Synthetic Data:

```
##########RL GBM classifier#############
=== Performance Comparison ===
RL-GBM => AUC: 0.7523, KS: 0.3686, ACC: 0.6886
XGBoost => AUC: 0.7512, KS: 0.3689, ACC: 0.6898
LightGBM => AUC: 0.7510, KS: 0.3667, ACC: 0.6883
AdaBoost => AUC: 0.7506, KS: 0.3709, ACC: 0.6880
RandomForest => AUC: 0.7492, KS: 0.3653, ACC: 0.6856
```

Predicting Above-Median Used Car Prices Using Kaggle Dataset:

```
=== RL-GBM Classifier ===
AUC: 0.7749, KS: 0.4206, ACC: 0.7083
XGBoost => AUC: 0.7763, KS: 0.4272, ACC: 0.7125
AdaBoost => AUC: 0.7621, KS: 0.4091, ACC: 0.7040
RandomForest => AUC: 0.7669, KS: 0.4099, ACC: 0.6997
LightGBM => AUC: 0.7755, KS: 0.4257, ACC: 0.7107
```

Summary of RL-Enhanced GBM Classifier Experiments

In two classification experiments — predicting synthetic customer purchase decisions and classifying used car prices from a *Kaggle* dataset — the Reinforcement Learning (*RL*)-enhanced *GBM* classifier showed competitive but mixed results. It performed similarly to *XGBoost* and *LightGBM*, slightly better in some metrics (e.g., A*UC* for synthetic data), but generally did not exceed these advanced models significantly. However, it consistently

outperformed traditional classifiers like *AdaBoost* and *Random Forest*, demonstrating its reliability as a powerful modeling option.

Compared to the earlier *RL*-enhanced *GBM* regression experiments — which clearly surpassed standard methods such as *XGBoost, LightGBM*, and Random Forest — the classifier version didn't show a similar level of superiority. This gap might exist because classification problems involve less direct gradients or noisier signals for the *RL* mechanism, making it harder for *RL* to effectively enhance the internal boosting process.

I think future improvements could focus on fine-tuning the *RL* policy, maybe by creating custom reward functions designed just for classification. Also, building smarter decision trees that work better with RL-guided splits could boost the results even more.

Final Thoughts

In this paper, I introduced an innovative approach combining Reinforcement Learning (*RL*) with Gradient Boosting Machines (*GBM*) to enhance predictive performance in both regression and classification tasks. Specifically, I developed the *RL-GBM* regression method, which dynamically adjusts gradient multipliers to minimize prediction errors, and the *RL-GBM classifier*, optimized for classification metrics like *AUC* and *KS*. To demonstrate their effectiveness, I applied these methods to real-world datasets from *Kaggle* (used car price prediction) and synthetic marketing data (purchase prediction), comparing their performance against strong models like *XGBoost, LightGBM, AdaBoost*, and Random Forest.

Overall, the *RL-GBM* regression method showed impressive performance gains, outperforming established algorithms consistently. The regression approach benefits significantly from *RL* due to the continuous nature of its target, allowing more precise gradient adjustments. On the other hand, the classifier, although competitive and generally surpassing Random Forest and AdaBoost, did not consistently outperform *XGBoost* or *LightGBM*. This suggests classification boundaries pose unique challenges, indicating room for future improvement.

Looking ahead, further research might explore more advanced *RL* strategies, refined reward functions, and custom tree-growing techniques specifically optimized for classification tasks. By deepening the integration of *RL* into boosting algorithms, we could unlock even stronger and more adaptable predictive models.

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Boosting Xgboost Lightgbm Reinforcement Learning Python

Responses (3)

Alex Mylnikov

What are your thoughts?

Matthias Wiedemann
Apr 6

Sounds Like a good idea to regulate over fitting

3 Reply

Salman Ahmed
Apr 6

hi i did remeber this approach was talked in early days when this technique was very outlet to improve gbm but it did not pickup much after that

7 Reply

Salvatore Raieli
Apr 5

It is very interesting, it is a cool hybrid approach, it is published? arxiv maybe?

8 1 reply Reply

More from Shenggang Li and Towards AI

In Towards AI by Shenggang Li

Reinforcement Learning for Business Optimization: A Genetic...

Applying PPO and Genetic Algorithms to Dynamic Pricing in Competitive Markets

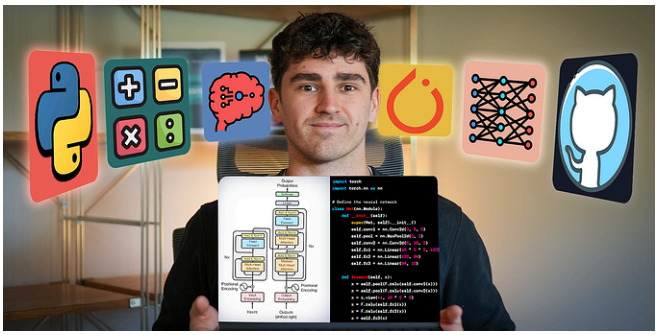
Mar 17 188 3

In Towards AI by Gao Dalie (高達烈)

Gemma 3 + MistralOCR + RAG Just Revolutionized Agent OCR Forever

Not a Month Ago, I made a video about Ollama-OCR. Many of you like this video

Mar 24 597 7



In Towards AI by Boris Meinardus

How I'd learn ML in 2025 (if I could start over)

All you need to learn ML in 2025 is a laptop and a list of the steps you must take.

Jan 2 1.7K 48

In Towards AI by Shenggang Li

Graph Neural Networks: Unlocking the Power of Relationships in...

Exploring the Concepts, Types, and Real-World Applications of GNNs in Feature...

Jan 11 310 3

See all from Shenggang Li

See all from Towards AI

Recommended from Medium

In Learn AI for Profit by Nipuna Maduranga

You Can Make Money With AI Without Quitting Your Job

I'm doing it, 2 hours a day

Mar 24 6.7K 303

Valeriy Manokhin, PhD, MBA, CQF

Predicting Full Probability Distributions with Conformal...

Introduction

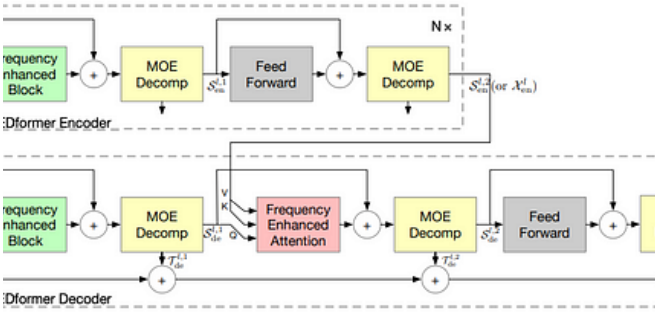
Mar 28 345 2

In Psyc Digest by Alessia Fransisca

The 1-Minute Introduction That Makes People Remember You...

A Behavioral Scientist's Trick to Hack the "Halo Effect"

Apr 12 20K 468

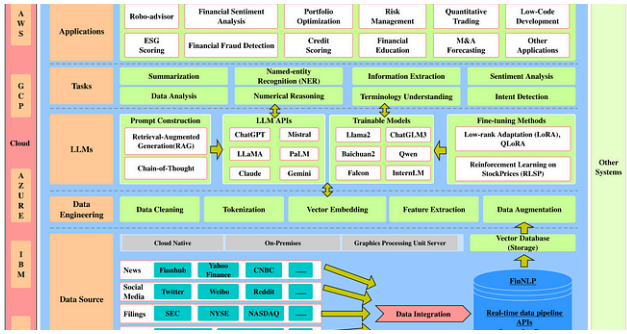


Dong-Keon Kim

FEDformer: Unleashing the Power of Frequency in Time Series...

A Deep Dive into Frequency Enhanced Decomposed Transformers for Long-Term...

Apr 13 21



In AI monks.io by JIN

FinGPT: The Future of Financial Analysis—Revolutionizing Marke...

Discover how FinGPT is disrupting traditional financial tools like Bloomberg Terminal,...

In Pythonic AF by Aysha R

I Tried Writing Python in VS Code and PyCharm—Here's What I...

One felt like a smart coding companion. The other felt like assembling IKEA furniture...

Feb 16 1K 21 Apr 15 879 71

See more recommendations