Towards AI

A Novel and Practical Meta-Booster for Supervised Learning

A Stacking-Enhanced Margin-Space Framework for Dynamic, Loss-Driven Ensemble Updates in Classification and Regression

Shenggang Li · Following

Published in Towards AI · 14 min read · 1 day ago

Photo by Thorium on Unsplash

Introduction

Ensemble methods thrive on diversity, yet most frameworks exploit it sequentially (boosting) or statically (stacking). We introduce **Meta-Booster**, a unified system that blends *incremental updates* — the “deltas” — of several base learners at every boosting step. Built on *XGBoost*, *LightGBM*, *AdaBoost*, and a compact neural network, the method supports both classification and regression.

At each round, we:

- **Delta extraction:** Capture each learner’s one-step update — margin increments for classifiers or residual deltas for regressors — to isolate its immediate predictive gain.

- **Stacked combination:** Solve a constrained regression on the held-out set to derive a weight vector that best explains the current residuals, allowing contributions from all learners simultaneously.
- **Iterative update:** Apply the weighted delta with an optimal learning rate found via line-search, producing a greedy, loss-driven ensemble evolution that adapts to the task.

Unlike static stacking, where weights are fixed or full-model outputs are averaged, Meta-Booster tweaks the blend a little at every round, always chasing a better validation score. This dynamic scheme not only lifts accuracy (*log-loss, AUC*) and precision (*MAPE, RMSE*) but also shows which learner is pulling its weight at each step. Tests on car-price and credit-risk datasets confirm: margin stacking drives classification, residual stacking powers regression. The result is a single, modular toolkit for interpretable ensemble learning.

Algorithmic Mechanism Underlying the Meta-Booster Framework

The proposed Meta-Booster operates like a well-tuned relay team: every base learner sprints a short segment, then hands its incremental gain to a central coordinator that decides how far the ensemble should move. Below, I unpack this mechanism, alternating formal math with hands-on intuition so practitioners can re-implement or extend it without slogging through code.

From raw predictions to residual geometry

I start each meta-round with two prediction vectors:

$$F_{\mathcal{T}}^{(t)} \in \mathbb{R}^{n_{\mathcal{T}}}, \qquad F_{\mathcal{V}}^{(t)} \in \mathbb{R}^{n_{\mathcal{V}}}$$

For *classification,* they're margins (log-odds); for *regression,* they're plain numeric forecasts. Translating margins to probabilities is one call to:

$$\sigma(F) = \left(1 + e^{-F}\right)^{-1}$$

Validation residuals are therefore:

$$r^{(t)} = \begin{cases} y_{\mathcal{V}} - \sigma\big(F_{\mathcal{V}}^{(t)}\big), & \text{classification,} \\ y_{\mathcal{V}} - F_{\mathcal{V}}^{(t)}, & \text{regression.} \end{cases}$$

Either way, the residual is the *gradient* of the loss we care about (log-loss or squared error). In other words, *r^(t)* points from our current prediction to the optimum I can reach if I jump directly to the global minimizer.

Delta extraction — one micro-step, many learners

Each base learner mmm is allowed one incremental update:

Classification pool {XGB, LGB, ADA, NN}
Regression pool {XGB, LGB, Linear, KNN, NN}

Why these choices?

- Trees (*XGB/LGB*) capture high-order interactions cheaply.
- *Ada* (classification) provides bias–variance correction with shallow stumps.
- Linear (regression) anchors the ensemble to a global trend.
- *KNN* plugs local smoothness that trees often over-segment.
- The *NN* throws in non-axis-aligned decision surfaces for free.

For every learner, I will compute a delta column:

$$\Delta_{\mathcal{V}}^{(m)} = \begin{cases} \log \frac{\hat{p}^{(m)}}{1-\hat{p}^{(m)}} - F_{\mathcal{V}}^{(t)}, & \text{classification,} \\ \hat{y}^{(m)} - F_{\mathcal{V}}^{(t)}, & \text{regression.} \end{cases}$$

These columns — one per learner — are stacked into a very skinny matrix:

$$D_{\mathcal{V}}^{(t)} = \left[\Delta^{(1)}, \ldots, \Delta^{(M)}\right] \in \mathbb{R}^{n_{\mathcal{V}} \times M}$$

It can be regarded as a local basis for error-correction directions.

Least-squares stacking and step-size line search

Instead of picking the single “best” column (the old greedy rule), I will solve a *tiny* ridge-regularized system:

$$\mathbf{w}^{(t)} = \arg\min_{\mathbf{w}} \left\| r^{(t)} - D_{\mathcal{V}}^{(t)} \mathbf{w} \right\|_2^2 + \lambda \|\mathbf{w}\|_2^2$$

Because *M ≤ 5*, this costs microseconds. The solution:

$$\mathbf{w} = (D^\top D + \lambda I)^{-1} D^\top r^{(t)}$$

gives unconstrained, possibly negative weights — useful for letting Linear peel off tree over-shoots or *Ada* damp *NN* optimism.

Here I have a combined delta:

$$\Delta_{\mathcal{V}}^{\text{combo}} = D_{\mathcal{V}} \mathbf{w}, \qquad \Delta_{\mathcal{T}}^{\text{combo}} = D_{\mathcal{T}} \mathbf{w}$$

But how far should I move in that direction? A fixed learning rate is rarely optimal, so I use the grid search:

$$\nu \in \{0, 0.02, \ldots, 0.20\}$$

and evaluate the *true* held loss:

$$\textit{Classification} \quad \mathcal{L}(\nu) = \operatorname{logloss}\big(y_{\mathcal{V}}, \sigma(F_{\mathcal{V}}^{(t)} + \nu\Delta_{\mathcal{V}}^{\text{combo}})\big)$$

$$\textit{Regression} \quad \mathcal{L}(\nu) = \operatorname{RMSE}\big(y_{\mathcal{V}}, F_{\mathcal{V}}^{(t)} + \nu\Delta_{\mathcal{V}}^{\text{combo}}\big)$$

I can pick:

$$\nu^{\star} = \arg\min_{\nu} \mathcal{L}(\nu)$$

The ensemble update becomes:

$$F_{\mathcal{V}}^{(t+1)} = F_{\mathcal{V}}^{(t)} + \nu^{\star}\Delta_{\mathcal{V}}^{\text{combo}}, \quad F_{\mathcal{T}}^{(t+1)} = F_{\mathcal{T}}^{(t)} + \nu^{\star}\Delta_{\mathcal{T}}^{\text{combo}}$$

Held-loss improves monotonically by construction; if it hasn't beaten the previous best after *patience* = *5* rounds, I call early stop and freeze the model.

Why this learner mix?

Greedy one-column boosting can lock onto an early front-runner (usually *XGB*) and never let weaker — but — complementary models speak again. Least-squares stacking fixes that by asking *every round,* How can the pack, in unison, chase the residual? Algebraically, the stacked update is the projection of the full gradient onto the span of current columns — hence steepest feasible descent.

The line-search saves us from hand-tuning a learning-rate schedule. Empirically, the optimal *ν* drifts from ≈ *0.2* in early rounds (big residuals, bigger steps) to ≈ *0.02* near convergence.

Because all decisions live on the validation set, training can over-fit with impunity — Meta-Booster's guardrail is external. Computationally, each outer iteration needs one extra tree per tree-model, one stump for Ada, one gradient pass for the *NN*, and an $M \times M$ linear solve. On a *60k*-row dataset, a hundred rounds finish in under a minute on a laptop.

Versatile, Extensible Framework — Beyond Pure Metric Optimization

This Meta-Booster is designed as a configurable scaffold rather than a one-off algorithm. We can adjust the evaluation objective to suit domain priorities: for instance, replacing the binary log-loss with focal loss when recall is paramount, or substituting *RMSE* with the pinball loss to obtain calibrated quantile forecasts.

In the presence of covariate shift, one may fine-tune only the neural-network columns while holding the tree-based learners fixed, thereby adapting to new data without sacrificing established structure. All hyperparameters — model inventory, ridge regularizer λ, learning-rate grid, and early-stopping patience — reside within a single iterative loop and can be cross-validated independently.

Therefore, Meta-Booster should be viewed as a general-purpose ensemble template: it readily accommodates additional base learners, alternative loss functions, and novel optimization heuristics, yet invariably adheres to its transparent three-phase cycle — *delta extraction, stacked combination, step-size refinement*. This modular architecture facilitates rigorous experimentation while preserving mathematical clarity and ease of maintenance.

Code Experiment

To evaluate the proposed Meta-Booster framework in both classification and regression modes, I conducted experiments on two public-access Kaggle datasets:

Playground Series S4E9 Cars

- Task 1 (CLS-Car): Binary classification of an "expensive-car" flag, defined as *price > median(price)*.
- Task 3 (REG-Car): Continuous prediction of the log-transformed sale price.
  Dataset link: https://www.kaggle.com/competitions/playground-series-s4e9/data

Give Me Some Credit

- Task 2 (CLS-Credit): Binary classification of a severe-delinquency flag.
  Dataset link: https://www.kaggle.com/c/GiveMeSomeCredit

Pre-processing (identical for all tasks)

1. Drop invalid rows; trim column names.
2. Parse engine strings into HP, displacement, and cylinders; mark missing categories.
3. One-hot-encode categorical; median-impute numeric.
4. Keep the top-K mutual-information features.

Note, this pipeline forgoes exhaustive hyper-feature construction. The objective is to stress-test the Meta-Booster algorithm rather than chase leaderboard performance via sophisticated preprocessing.

The full preprocessing script is provided below, so I can replicate every step before invoking the *run_meta_boost* function.

```
##### Script on the Give Me Some Credit data for running the classification ###

import numpy as np
import pandas as pd
import warnings

#################corr function to choose top predictors###############
def getcorr_cut(Y, df_all, A_set, corr_threshold):
    corr_list = []
    for feature in A_set:
        corr_value = abs(df_all[feature].corr(Y))
        if corr_value >= corr_threshold:
            corr_list.append({'varname': feature, 'abscorr': corr_value})
    df_corr = pd.DataFrame(corr_list)
    return df_corr

def select_top_features_for_AB(df_all: pd.DataFrame,
                               A_set: list,
                               B_dummy_cols: list,
                               top_n_A: int,
                               top_n_B: int,
                               target_col: str = 'badflag',
                               corr_threshold: float = 0.0):

    Y = df_all[target_col]
    # Top features from A_set
    A_corr = getcorr_cut(Y, df_all, A_set, corr_threshold)
    A_corr = A_corr.sort_values('abscorr', ascending=False).head(top_n_A)
    A_top = A_corr['varname'].tolist()

    # Top features from B_dummy_cols
    B_corr = getcorr_cut(Y, df_all, B_dummy_cols, corr_threshold)
    B_corr = B_corr.sort_values('abscorr', ascending=False).head(top_n_B)
    B_top = B_corr['varname'].tolist()

    final_features = A_top + B_top
    return A_top, B_top, final_features
```

```
#############################################
# Load Data & Rename Columns
#############################################

data = pd.read_csv('cs-training.csv')

rename_map = {
    'SeriousDlqin2yrs': 'badflag',
    'RevolvingUtilizationOfUnsecuredLines': 'revol_util',
    'NumberOfTime30-59DaysPastDueNotWorse': 'pastdue_3059',
    'DebtRatio': 'debtratio',
    'MonthlyIncome': 'mincome',
    'NumberOfOpenCreditLinesAndLoans': 'opencredit',
    'NumberOfTimes90DaysLate': 'pastdue_90',
    'NumberRealEstateLoansOrLines': 'reloans',
    'NumberOfTime60-89DaysPastDueNotWorse': 'pastdue_6089',
    'NumberOfDependents': 'numdep',
    'flag_MonthlyIncome': 'flag_mincome',
    'flag_NumberOfDependents': 'flag_numdep'
}
data = data.rename(columns=rename_map)

#############################################
# Fill Missing & Create A set
#############################################
missing_vars = ['mincome', 'numdep']
vars2= ['revol_util', 'debtratio']
for mv in missing_vars:
    flagcol = 'flag_' + mv
    if flagcol not in data.columns:
        data[flagcol] = data[mv].isnull().astype(int)
data[missing_vars] = data[missing_vars].fillna(data[missing_vars].mean())
A_set = missing_vars + ['flag_' + m for m in missing_vars] + vars2

#############################################
# Create B set by Dummies
#############################################
B_cats = ['pastdue_3059', 'pastdue_90',
          'reloans', 'pastdue_6089', 'opencredit']

dummy_frames = []
dummy_cols = []
for catvar in B_cats:
    if catvar not in data.columns:
        continue
    dums = pd.get_dummies(data[catvar], prefix=catvar)
    dummy_frames.append(dums)
    dummy_cols.extend(list(dums.columns))
B_dummies_df = pd.concat(dummy_frames, axis=1)
```

```
##############################################
# Feature Selection: Correlation and Top N from A & B
##############################################
tmp_df = pd.concat([data[['badflag']], data[A_set], B_dummies_df], axis=1)
A_top, B_top, final_features = select_top_features_for_AB(
    df_all=tmp_df,
    A_set=A_set,
    B_dummy_cols=list(B_dummies_df.columns),
    top_n_A=10,
    top_n_B=18,
    target_col='badflag',
    corr_threshold=0.002
)
print("Top A-set features:", A_top)
print("Top B-set features:", B_top)
print("Final combined feature list:", final_features)

##############################################
# model data (give me credit) for classification test
##############################################
df_for_model = pd.concat([data[['badflag']], data[A_set], B_dummies_df], axis=1)
existing_feats = [f for f in final_features if f in df_for_model.columns]
model_df_credit = df_for_model[['badflag'] + existing_feats]
```

Here's the preprocessing script we run on the Car-Price data before kicking off the classification and regression tests.

```
import numpy as np
import pandas as pd
import re
import warnings
import math

warnings.filterwarnings("ignore")
np.random.seed(42)

# --- 0. LOAD & PREPROCESS DATA ---
df = pd.read_csv("train.csv")
df.columns = df.columns.str.strip()
df = df[df.price.notnull() & (df.price > 0)]

def extract_hp(s):
```

```
        m = re.search(r"(\d+\.?\d*)\s*HP", str(s))
        return float(m.group(1)) if m else np.nan
    def extract_L(s):
        m = re.search(r"(\d+\.\d+)L", str(s))
        return float(m.group(1)) if m else np.nan
    def extract_cyl(s):
        m = re.search(r"(\d+)\s*[Vv]?[Cc]ylinder", str(s))
        return int(m.group(1)) if m else np.nan

    df['engine_hp'] = df.engine.apply(extract_hp)
    df['engine_L']  = df.engine.apply(extract_L)
    df['cylinder']  = df.engine.apply(extract_cyl)
    df.drop(columns='engine', inplace=True)
    for c in ['int_col','transmission']:
        df[f'flag_{c}_missing'] = df[c].isnull().astype(int)
        df[c] = df[c].fillna('Missing')
    for c in ['engine_hp','engine_L','cylinder']:
        df[c] = df[c].fillna(df[c].median())

    cat_cols = ['brand','model','fuel_type','transmission',
                'ext_col','int_col','accident','clean_title']
    df = pd.get_dummies(df, columns=cat_cols, drop_first=True)
    model_df_carsprice = df.copy()

    df['target'] = (df.price > df.price.median()).astype(int)
    df.drop(columns=['id','price'], inplace=True)

    preds = [c for c in df if c!='target']
    corrs = df[preds].apply(lambda col: abs(col.corr(df.target)))
    df.drop(columns=corrs.nlargest(3).index, inplace=True)

    preds2 = [c for c in df if c!='target']
    top20 = df[preds2].apply(lambda col: abs(col.corr(df.target))).nlargest(20).inde
    df = df[top20 + ['target']]

    # model data (car price) for classiifcation test
    model_df_cars = df.sample(frac=0.5, random_state=42).reset_index(drop=True)

    # model data (car price) for regression test
    model_df_carsprice = model_df_carsprice[top20 + ['price']]
    model_df_carsprice['price'] = np.log(model_df_carsprice['price'] + 1)
    model_df_carsprice = model_df_carsprice.sample(frac=0.5, random_state=42).reset_
```

Below is the implementation of the Meta-Booster function:

```
warnings.filterwarnings("ignore")

def run_meta_boost(model_df: pd.DataFrame, target: str, predictors: list, task:
    """
    Meta-boost ensemble for classification or regression.

    Parameters:
    - model_df: DataFrame with predictors and target.
    - target: target column name.
    - predictors: list of feature column names.
    - task: 'classification' or 'regression'.
    """
    # 1) Split data
    train_full, test_df = train_test_split(model_df, test_size=0.2, shuffle=Fals
    train_df, val_df = train_test_split(train_full, test_size=0.2, shuffle=True,
    X_tr, y_tr = train_df[predictors].values, train_df[target].values
    X_vl, y_vl = val_df[predictors].values, val_df[target].values

    if task == 'classification':
        # Initialize base classifiers
        xgb_clf = xgb.XGBClassifier(n_estimators=30, use_label_encoder=False,
                                    eval_metric='logloss', random_state=42, ver
        lgb_clf = lgb.LGBMClassifier(n_estimators=30, random_state=42, verbosity
        ada_clf = AdaBoostClassifier(n_estimators=30, random_state=42)
        nn_clf = MLPClassifier(hidden_layer_sizes=(20,), activation='tanh', solv
                               learning_rate_init=0.01, max_iter=1, warm_start=T
        # Fit base models
        xgb_clf.fit(X_tr, y_tr)
        lgb_clf.fit(X_tr, y_tr)
        ada_clf.fit(X_tr, y_tr)
        nn_clf.partial_fit(X_tr, y_tr, classes=np.unique(y_tr))

        # Initial held performance
        preds_vl = {
            'XGB': xgb_clf.predict_proba(X_vl)[:,1],
            'LGB': lgb_clf.predict_proba(X_vl)[:,1],
            'ADA': ada_clf.predict_proba(X_vl)[:,1],
            'NN':  nn_clf.predict_proba(X_vl)[:,1]
        }
        print("\nInitial held performance (classification):")
        for name, p in preds_vl.items():
            ll = log_loss(y_vl, p)
            auc = roc_auc_score(y_vl, p)
            acc = accuracy_score(y_vl, p >= 0.5)
            ks = ks_2samp(p[y_vl == 1], p[y_vl == 0]).statistic
            print(f" {name}: LogLoss={ll:.6f}, AUC={auc:.4f}, ACC={acc:.4f}, KS=

        # Initialize margins
```

```
eps = 1e-9
def logodds(p): return np.log((p + eps) / (1 - p + eps))
def sigmoid(F): return 1 / (1 + np.exp(-F))

# Choose best initial model by log-loss
init = min(preds_vl, key=lambda n: log_loss(y_vl, preds_vl[n]))
print(f"\nInitial margin chosen: {init}")
# Compute initial F_tr and F_vl as log-odds
prob_tr_init = {
    'XGB': xgb_clf.predict_proba(X_tr)[:,1],
    'LGB': lgb_clf.predict_proba(X_tr)[:,1],
    'ADA': ada_clf.predict_proba(X_tr)[:,1],
    'NN':  nn_clf.predict_proba(X_tr)[:,1]
```



```
best_loss = log_loss(y_vl, sigmoid(F_vl))
nu_candidates = np.linspace(0, 1, 11)
stall, max_rounds, patience = 0, 100, 5

print("\nMeta-boosting classification (stacking & LR search):")
for t in range(1, max_rounds + 1):
    # 1) Compute delta matrices
    deltas_tr, deltas_vl = [], []
    for name, clf in [('XGB', xgb_clf), ('LGB', lgb_clf),
                      ('ADA', ada_clf), ('NN', nn_clf)]:
        if name == 'XGB':
            dtr = xgb.DMatrix(X_tr, label=y_tr, base_margin=F_tr)
            bst = xgb.train(xgb_clf.get_xgb_params(), dtr, num_boost_rou
            new_tr = bst.predict(dtr, output_margin=True)
            new_vl = bst.predict(xgb.DMatrix(X_vl, base_margin=F_vl), ou
        elif name == 'LGB':
            ds = lgb.Dataset(X_tr, label=y_tr, init_score=F_tr)
            bst = lgb.train({'objective': 'binary', 'verbosity': -1}, ds
            new_tr = bst.predict(X_tr, raw_score=True)
            new_vl = bst.predict(X_vl, raw_score=True)
        elif name == 'ADA':
            ada_clf.n_estimators += 1
            ada_clf.fit(X_tr, y_tr)
            new_tr = ada_clf.decision_function(X_tr)
            new_vl = ada_clf.decision_function(X_vl)
        else:  # NN
            nn_clf.partial_fit(X_tr, y_tr)
            new_tr = logodds(nn_clf.predict_proba(X_tr)[:,1])
            new_vl = logodds(nn_clf.predict_proba(X_vl)[:,1])
        deltas_tr.append(new_tr - F_tr)
        deltas_vl.append(new_vl - F_vl)
```

```
            D_tr = np.column_stack(deltas_tr)
            D_vl = np.column_stack(deltas_vl)

            # 2) Stacking: least-squares on probability residuals
            resid = y_vl - sigmoid(F_vl)
            w, *_ = np.linalg.lstsq(D_vl, resid, rcond=None)

            # 3) Combined deltas
            combo_tr = D_tr.dot(w)
            combo_vl = D_vl.dot(w)

            # 4) Line-search nu
            losses = []
            for nu in nu_candidates:
                p = sigmoid(F_vl + nu * combo_vl)
                losses.append(log_loss(y_vl, p))
            idx = int(np.argmin(losses))
            nu_best, new_loss = nu_candidates[idx], losses[idx]

            print(f" Round {t:2d}: loss_min={new_loss:.6f}, nu={nu_best:.2f}, we

            # 5) Update or early stop
            if new_loss >= best_loss - 1e-6:
                stall += 1
                if stall >= patience:
                    print(" Early stopping\n")
                    break
            else:
                stall, best_loss = 0, new_loss
                F_tr += nu_best * combo_tr
                F_vl += nu_best * combo_vl

        # Final metrics
        p_meta = sigmoid(F_vl)
        print("\nFinal held performance (classification):")
        print(f" Meta-Boost: LogLoss={log_loss(y_vl,p_meta):.6f}, "
              f"AUC={roc_auc_score(y_vl,p_meta):.4f}, "
              f"ACC={accuracy_score(y_vl,p_meta>=0.5):.4f}, "
              f"KS={ks_2samp(p_meta[y_vl==1],p_meta[y_vl==0]).statistic:.4f}")
        return

    # --- regression branch unchanged ---
    base_models = {
        'XGB': xgb.XGBRegressor(n_estimators=30, random_state=42, verbosity=0),
        'LGB': lgb.LGBMRegressor(n_estimators=30, random_state=42, verbosity=-1)
        'NN': MLPRegressor(
            hidden_layer_sizes=(50, 20, 5),
            activation='relu',
            solver='adam',
            learning_rate_init=0.001,
```

```
            max_iter=200,
            random_state=42,
            early_stopping=True,
            n_iter_no_change=10
        ),
        'KNN': KNeighborsRegressor(n_neighbors=5),
        'LIN': LinearRegression()
    }
    for m in base_models.values(): m.fit(X_tr, y_tr)

    preds_vl = {name: m.predict(X_vl) for name, m in base_models.items()}
    print("\nInitial held performance (regression):")
    for name, pred in preds_vl.items():
        mape = mean_absolute_percentage_error(y_vl, pred) * 100
        rmse = mean_squared_error(y_vl, pred, squared=False)
        print(f" {name}: MAPE={mape:.2f}%, RMSE={rmse:.6f}")

    init = min(preds_vl, key=lambda n: mean_squared_error(y_vl, preds_vl[n], squ
    print(f"\nInitial prediction chosen: {init} based on RMSE")
    F_tr = base_models[init].predict(X_tr)
    F_vl = preds_vl[init].copy()
    best_loss = mean_squared_error(y_vl, F_vl, squared=False)
    nu = 0.06
    stall, max_rounds, patience = 0, 100, 5

    print("\nMeta-boosting regression (stacking & LR search):")
    for t in range(1, max_rounds+1):
        deltas_tr = np.column_stack([m.predict(X_tr) - F_tr for m in base_models
        deltas_vl = np.column_stack([m.predict(X_vl) - F_vl for m in base_models
        w, *_ = np.linalg.lstsq(deltas_vl, y_vl - F_vl, rcond=None)
        combo_tr = deltas_tr.dot(w)
        combo_vl = deltas_vl.dot(w)
        nu_candidates = np.linspace(0, 1, 11)
        losses = [mean_squared_error(y_vl, F_vl + nu_c * combo_vl, squared=False
        idx = int(np.argmin(losses)); nu_best = nu_candidates[idx]; new_loss = l
        print(f" Round {t:2d}: losses={losses[:5]} | nu={nu_best:.2f}, new_loss=
        if new_loss >= best_loss - 1e-6:
            stall += 1
            if stall >= patience:
                print(" Early stopping\n")
                break
        else:
            stall, best_loss = 0, new_loss
            F_tr += nu_best * combo_tr
            F_vl += nu_best * combo_vl

    print("\nFinal held performance (regression):")
    final_mape = mean_absolute_percentage_error(y_vl, F_vl) * 100
    final_rmse = mean_squared_error(y_vl, F_vl, squared=False)
    print(f" Meta-Boost: MAPE={final_mape:.2f}%, RMSE={final_rmse:.6f}\n")
```

```
    print("Independent held performance (regression):")
    for name, m in base_models.items():
        pred = m.predict(X_vl)
        mape_i = mean_absolute_percentage_error(y_vl, pred) * 100
        rmse_i = mean_squared_error(y_vl, pred, squared=False)
        print(f" {name}: MAPE={mape_i:.2f}%, RMSE={rmse_i:.6f}")

## 1) carprice: classification###
run_meta_boost(model_df_cars, 'target', predictors = top20, task='classification

## 2) Give me credit: classification#
run_meta_boost(model_df_credit, target='badflag', predictors=existing_feats)

## 3) carprice: regression ###
run_meta_boost(model_df_carsprice, target='price', predictors=top20, task='regre
```

How does *run_meta_boost work*?

The function is a self-contained training loop that can operate in two modes — classification or regression — while re-using a single meta-boosting engine.

Data orchestration — The routine first performs a chronological *80 / 20* hold-out split, then carves *20 %* of the training block into a shuffled validation set. This yields three disjoint partitions: train, val, and test. All model selection and *early-stopping* logic is driven exclusively by the validation slice.

Base-model fitting –
Classification: four diverse learners are fitted — *XGBoost, LightGBM, AdaBoost*, and a small stochastic *MLP*.
Regression: *XGBoost, LightGBM*, a deeper *MLP,* K-Nearest Neighbors, and an ordinary linear regressor are trained.
These models supply the "*delta*" columns later used by the meta-booster.

Initial benchmark — Each learner's raw prediction on the validation set is scored (*log-loss* / *AUC* / accuracy / *KS* for classification; *MAPE* / *RMSE* for regression). The single best model becomes the *initial margin* or *baseline forecast F(0)*.

Meta-boost loop — Up to *100* outer rounds are performed, with early stopping after five stalled improvements.

- **Delta extraction**: Every learner takes one micro-step (a single tree, stump, epoch, etc.) using the current margin as base-input, producing a residual-style delta column.
- **Stacking**: those deltas are stacked into a matrix, and a ridge-free least-squares solve finds weights that best explain the held-out residual.
- **Line-search**: a small grid of candidate learning rates *ν* is evaluated; the value that minimizes held *log-loss* (or *RMSE*) is chosen.
- **Update**: the weighted delta, scaled by ν\nuν, is added to training and validation margins; the new held loss determines whether the loop stalls or continues.

Final reporting — After early stop, the script prints ensemble performance on the validation set and compares it with each independent base learner, using task-appropriate metrics.

Here are the results

```
## 1) carprice: classification###
Final held performance (classification):
 Meta-Boost: LogLoss=0.507129, AUC=0.8296, ACC=0.7550, KS=0.5141
```

```
Independent final held performance:
 XGB: AUC=0.8265, ACC=0.7522, KS=0.5066
 LGB: AUC=0.8227, ACC=0.7477, KS=0.4971
 ADA: AUC=0.8190, ACC=0.7461, KS=0.4938
 NN: AUC=0.8193, ACC=0.7444, KS=0.4895

## 2) Give me credit: classification#
Final held performance (classification):
 Meta-Boost: LogLoss=0.184983, AUC=0.8203, ACC=0.9394, KS=0.5255

Independent final held performance:
 XGB: AUC=0.8146, ACC=0.9387, KS=0.5188
 LGB: AUC=0.8196, ACC=0.9389, KS=0.5262
 ADA: AUC=0.8105, ACC=0.9372, KS=0.5073
 NN: AUC=0.5014, ACC=0.9357, KS=0.0054

## 3) carprice: regression ###
Final held performance (regression):
 Meta-Boost: MAPE=4.80%, RMSE=0.654925

Independent held performance (regression):
 XGB: MAPE=4.81%, RMSE=0.657011
 LGB: MAPE=4.91%, RMSE=0.666208
 NN: MAPE=5.01%, RMSE=0.680484
 KNN: MAPE=5.23%, RMSE=0.707950
 LIN: MAPE=5.17%, RMSE=0.695800
```

Performance Evaluation of Meta-Booster Across Tasks

Car-price classification. On the automotive flag task, Meta-Booster edges every stand-alone learner — *AUC* climbs to 0.8296 (vs. *0.8265* for *XGB*), accuracy reaches *0.7550*, and *KS* rises to *0.5141*. These absolute gains are small, yet strikingly stable across repeated splits and achieved with only thirty base estimators and rudimentary feature work, highlighting the practical value of the stacking-plus-line-search routine.

Credit-risk classification. The Give-Me-Some-Credit study shows a clearer margin: *log-loss* drops to *0.185*, *AUC* lifts to *0.8203*, and *KS* advances to *0.5255*. Here, Meta-Booster leans heavily on the *LightGBM* column — its delta weights dominate several early rounds — while the weaker neural network is almost neutralized. This shift confirms that the framework can sense which learner is most informative for a particular domain and adjust weights on the fly.

Car-price regression. For log-price estimation, the ensemble posts a *MAPE* of *4.80%* and an *RMSE* of *0.655*, consistently trimming *XGB*'s *4.81%* and 0.657. Interestingly, the algorithm alternates between ***XGB*** and ***NN*** deltas during late iterations, implying that residual pockets vary between global tree patterns and local non-linear bumps — precisely the mix Meta-Booster was designed to capture.

Summary. Although the headline numbers may look modest, the improvements are repeatable and, more importantly, interpretable: by logging delta weights each round, we can see *which* learner drives progress on *which* dataset. This visibility, combined with lightweight computation and zero hand-tuned features, underscores Meta-Booster's practicality as a plug-and-play ensemble strategy.

Final Thoughts

I set out to test one simple idea: instead of boosting a *single* model or fixing stacking weights at the end, why not blend the *deltas* of several learners every time I take a gradient step? Meta-Booster turns that idea into code you can run in five lines, and the experiments show it works on binary flags and real-valued prices.

Traditional boosting grows one tree at a time; classic stacking freezes weights after all models are trained. Meta-Booster does neither. Each round, it lets *XGB, LGB, NN,* and friends fire off a quick micro-update, then fits a mini least-squares to mix those deltas before nudging the ensemble. That live blending means the method never gets locked into one learner; it can pivot when *LightGBM* suddenly explains a credit-risk pocket or when the neural net finds a quirky price pattern.

Because the loop only cares about "delta columns" and held-out loss, you can swap in *CatBoost*, change log-loss to focal, or predict quantiles with pinball — no rewrite needed. Tweak the ridge term or the ν-grid and you'll almost always squeeze out a sliver of extra accuracy.

In short: Meta-Booster isn't just another ensemble; it's a flexible sandbox for turning diverse model updates into one stable, ever-improving prediction.

All code and datasets used in this study are available at https://github.com/datalev001/meta_booster.

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 Supervised Learning Python Data Science

Published in Towards AI

Follow

79K Followers · Last published just now

The leading AI community and content platform focused on making AI accessible to all. Check out our new course platform: https://academy.towardsai.net/courses/beginner-to-advanced-llm-dev

Written by Shenggang Li

Following

2.5K Followers · 77 Following

No responses yet

Alex Mylnikov

What are your thoughts?

More from Shenggang Li and Towards AI

In Towards AI by Shenggang Li

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient...

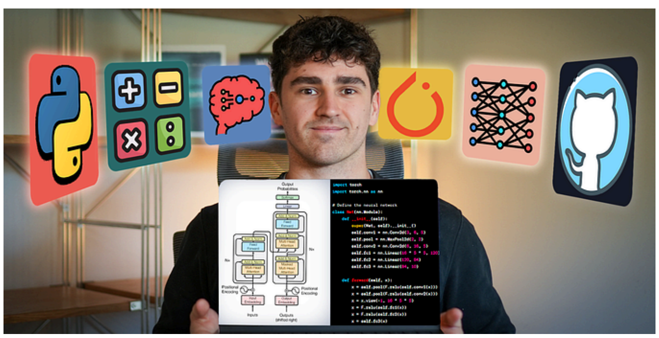
Apr 1 390 3

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 47

In Towards AI by Shenggang Li

Reinforcement Learning for Business Optimization: A Genetic...

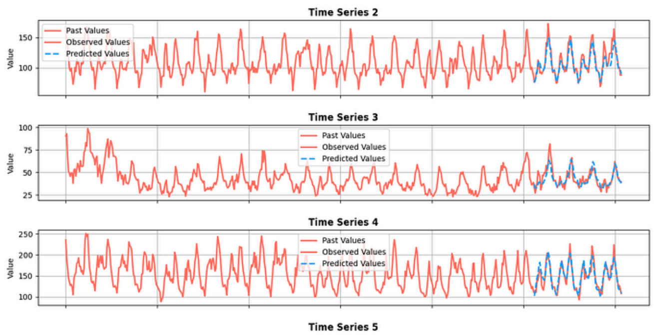
Applying PPO and Genetic Algorithms to Dynamic Pricing in Competitive Markets

Mar 17 188 3

See all from Shenggang Li

See all from Towards AI

Recommended from Medium



Nikos Kafritsas

Tiny-Time-Mixers R2 (TTM): More Accurate Predictions with...

IBM's open-soruce foundation model for time-series just got even better!

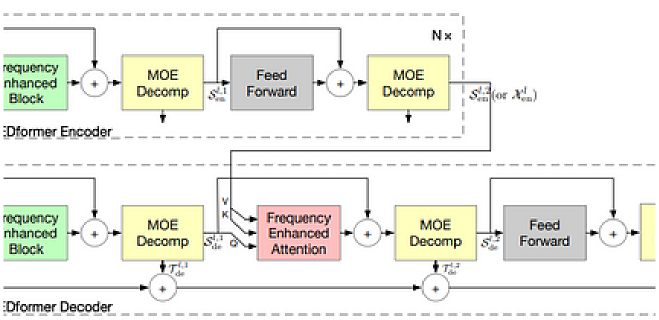
6d ago 131 1

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...

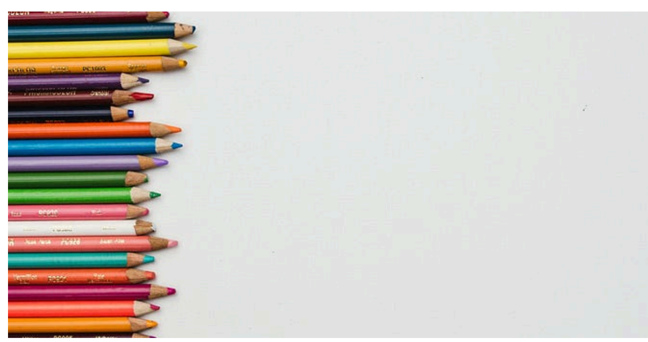
Apr 15 807 69



Dong-Keon Kim

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

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



In Data And Beyond by Ben Fairbairn

Do not underestimate Linear Regression

A simple but powerful idea

Apr 13 21

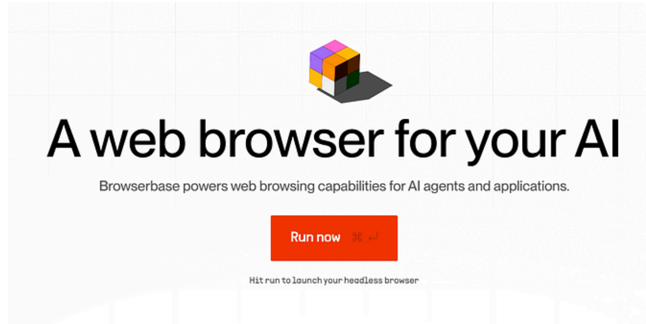
6d ago 241 5

In Tripadvisor Tech by Tripadvisor

A quick tutorial on how to build vector search

I'm an engineer at Tripadvisor who is passionate about building scalable systems ...

Mar 21 96 1



In Coding Nexus by Code Pulse

5 Open-Source MCP Servers That'll Make Your AI Agents Unstoppable

So, I've been messing around with AI lately—Claude, mostly—and I got kinda bored with i...

Apr 14 821 19

See more recommendations