Model Validation Suite: Overfitting Diagnostics

MSBA Capstone - MasterControl Lead Scoring (Synced with V7 Titan)

MSBA Capstone Group 3

2026-01-01

Executive Summary

Purpose: This validation suite answers the critical question sponsors will ask: *"How do we know your model won't fall apart in production?"*

This script performs rigorous overfitting diagnostics by comparing training performance to holdout test performance across multiple model architectures. The **Generalization Gap** (Training AUC - Test AUC) reveals which models have genuinely learned patterns versus which have memorized noise.

Key Metrics:

Metric	Interpretation	Threshold
Generalization Gap	Train AUC - Test AUC	< 0.05 = Good, > 0.10 = Concerning
ROC-AUC Slope	Performance consistency across thresholds	Steeper = Better calibration
Training AUC	Upper bound of model capability	Context for test performance
Test AUC	Real-world expected performance	Primary decision metric

---

Phase 1: Environment Setup

```
# ============================================================================
# PHASE 1: VALIDATION ENVIRONMENT
# ============================================================================

import subprocess
import sys

def install_if_missing(package_name, import_name=None, pip_name=None):
    """Install a package if not already available."""
    import_name = import_name or package_name.lower()
    pip_name = pip_name or import_name

    try:
        __import__(import_name)
        return True
    except ImportError:
        print(f"{package_name}: Not found. Installing...")
        try:
            subprocess.check_call(
                [sys.executable, "-m", "pip", "install", pip_name, "-q"],
                stdout=subprocess.DEVNULL,
                stderr=subprocess.DEVNULL
            )
            print(f"{package_name}: Installed successfully!")
            return True
        except subprocess.CalledProcessError:
            print(f"{package_name}: Installation failed.")
            return False

# Install dependencies
print("=" * 70)
print("MODEL VALIDATION SUITE: CHECKING DEPENDENCIES")
print("=" * 70)

install_if_missing("pandas")
install_if_missing("numpy")
install_if_missing("matplotlib")
install_if_missing("seaborn")
install_if_missing("scikit-learn", import_name="sklearn", pip_name="scikit-learn")
install_if_missing("pyprojroot", import_name="pyprojroot")
install_if_missing("CatBoost", import_name="catboost")
install_if_missing("XGBoost", import_name="xgboost")
install_if_missing("LightGBM", import_name="lightgbm")

print("=" * 70)

# Core imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import time
import re
from pathlib import Path
from datetime import datetime
from pyprojroot import here
from types import SimpleNamespace

warnings.filterwarnings('ignore')

# ML imports
from sklearn.model_selection import (
    train_test_split, StratifiedKFold, RandomizedSearchCV, cross_val_predict
)
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.base import BaseEstimator, ClassifierMixin, clone
```

```python
# Metrics
from sklearn.metrics import (
    roc_auc_score, roc_curve, precision_recall_curve, average_precision_score,
    classification_report, confusion_matrix, brier_score_loss, log_loss
)

# Models
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

# Boosting libraries
CATBOOST_AVAILABLE = False
try:
    from catboost import CatBoostClassifier as CatBoostRaw
    CATBOOST_AVAILABLE = True
    print("CatBoost: Ready")
except ImportError:
    print("CatBoost: Not available")

XGBOOST_AVAILABLE = False
try:
    from xgboost import XGBClassifier
    XGBOOST_AVAILABLE = True
    print("XGBoost: Ready")
except ImportError:
    print("XGBoost: Not available")

LIGHTGBM_AVAILABLE = False
try:
    from lightgbm import LGBMClassifier
    LIGHTGBM_AVAILABLE = True
    print("LightGBM: Ready")
except ImportError:
    print("LightGBM: Not available")

# ==============================================================================
# CATBOOST SKLEARN WRAPPER (from V7)
# ==============================================================================

if CATBOOST_AVAILABLE:
    class SklearnCatBoost(BaseEstimator, ClassifierMixin):
        """sklearn-compatible CatBoost wrapper for sklearn 1.6+"""
        _estimator_type = "classifier"

        def __init__(self, iterations=500, depth=6, learning_rate=0.1,
                     l2_leaf_reg=3, border_count=64, random_state=42,
                     verbose=0, thread_count=1):
            self.iterations = iterations
            self.depth = depth
            self.learning_rate = learning_rate
            self.l2_leaf_reg = l2_leaf_reg
            self.border_count = border_count
            self.random_state = random_state
            self.verbose = verbose
            self.thread_count = thread_count
            self._model = None

        def __sklearn_tags__(self):
            tags = SimpleNamespace()
            tags.estimator_type = "classifier"
            tags.classifier_tags = SimpleNamespace()
            tags.regressor_tags = None
            tags.transformer_tags = None
            tags.input_tags = SimpleNamespace(allow_nan=True, pairwise=False)
            tags.target_tags = SimpleNamespace(required_y=True)
            return tags

        def fit(self, X, y, **fit_params):
            self._model = CatBoostRaw(
                iterations=self.iterations, depth=self.depth,
                learning_rate=self.learning_rate, l2_leaf_reg=self.l2_leaf_reg,
                border_count=self.border_count, random_state=self.random_state,
                verbose=self.verbose, thread_count=self.thread_count,
                allow_writing_files=False
            )
            self._model.fit(X, y, **fit_params)
            self.classes_ = np.unique(y)
            return self

        def predict(self, X):
            return self._model.predict(X).flatten().astype(int)

        def predict_proba(self, X):
            return self._model.predict_proba(X)

        @property
        def feature_importances_(self):
            return self._model.get_feature_importance()

# Configuration
RANDOM_STATE = 42
CV_FOLDS = 5
TEST_SIZE = 0.20
COST_PER_CALL = 50
VALUE_PER_SQL = 6000

# Visual configuration
PROJECT_COLS = {
    'Success': '#00534B',
    'Failure': '#F05627',
    'Neutral': '#95a5a6',
    'Highlight': '#2980b9',
    'Premium': '#2ecc71',
    'Toxic': '#e74c3c'
}
```

```
sns.set_theme(style="whitegrid", context="talk")
plt.rcParams['figure.figsize'] = (14, 8)

# Path configuration
DATA_DIR = here("data")
OUTPUT_DIR = here("output")
CLEANED_DATA_PATH = here("output/Cleaned_QAL_Performance_for_MSBA.csv")
RAW_DATA_PATH = here("data/QAL Performance for MSBA.csv")

if CLEANED_DATA_PATH.exists():
    DATA_PATH = CLEANED_DATA_PATH
    print(f"\nUsing cleaned data: {CLEANED_DATA_PATH}")
else:
    DATA_PATH = RAW_DATA_PATH
    print(f"\nUsing raw data: {RAW_DATA_PATH}")

print("\n" + "=" * 70)
print("VALIDATION SUITE INITIALIZED")
print("Synced with V7 Titan Feature Engineering")
print("=" * 70)
```

```
======================================================================
MODEL VALIDATION SUITE: CHECKING DEPENDENCIES
======================================================================
======================================================================
CatBoost: Ready
XGBoost: Ready
LightGBM: Ready

Using cleaned data: C:\Users\thoma\Repos\MSBA-Capstone-MasterControl-GroupProject\output\Cleaned_QAL_Pe

======================================================================
VALIDATION SUITE INITIALIZED
Synced with V7 Titan Feature Engineering
======================================================================
```

Phase 2: V7 Titan Feature Engineering (SYNCED)

CRITICAL: This feature engineering MUST match the V7 Python pipeline exactly.

```
# ============================================================================
# PHASE 2: V7 TITAN FEATURE ENGINEERING (EXACT COPY FROM V7)
# ============================================================================

# ----------------------------------------------------------------------------
# V7 TITAN MAPPING DEFINITIONS
# ----------------------------------------------------------------------------

# 1. INTENT STRENGTH (Ordinal Encoding of Priority)
INTENT_STRENGTH_MAP = {
    'P1 - Website Pricing': 5,
    'P1 - Contact Us': 5,
    'P1 - Video Demo': 3,
    'P1 - Live Demo': 3,
    'P1 - Webinar Demo': 1,
    'No Priority': 1,
    'Priority 1': 2,
    'Priority 2': 0
}

# 2. CHANNEL EFFICIENCY (Tiered Lead Source Quality)
CHANNEL_TIER_MAP = {
    'Direct/Inbound': 'Premium',
    'SEO': 'Premium',
    'Referrals': 'Premium',
    'Online Ads': 'Standard',
    'Directory Listing': 'Standard',
    'Events': 'Standard',
    'Outbound Prospecting': 'Standard',
    'Email': 'Toxic',
    'External Demand Gen': 'Toxic'
}

CHANNEL_NUMERIC_MAP = {
    'Premium': 3,
    'Standard': 2,
    'Toxic': 1,
    'Unknown': 2
}

# 3. CAPITAL DENSITY (Industry-Weighted Budget Proxy)
INDUSTRY_BUDGET_MULTIPLIER = {
    'Pharma & BioTech': 3.0,
    'Blood & Biologics': 2.5,
    'Medical Device': 2.0,
    'Non-Life Science': 1.0,
    'Consumer Packaged Goods': 0.8
}

TIER_SIZE_MAP = {
    'Small': 50,
    'Medium': 500,
    'Large': 5000
}

# 4. HIGH-VALUE TITLE BIGRAMS
HIGH_VALUE_BIGRAMS = [
    'continuous improvement',
    'document control',
    'process engineer',
    'quality systems',
    'regulatory affairs',
    'quality assurance',
    'validation engineer',
    'compliance manager'
```

```
]

# 5. ROLE-PRODUCT ALIGNMENT
PRODUCT_ROLE_ALIGNMENT = {
    'Mx': ['Op', 'Mfg', 'Manuf', 'Production', 'Plant'],
    'Qx': ['Qual', 'QA', 'QC', 'Compliance', 'Validation']
}

# ---------------------------------------------------------------------------
# V7 TITAN FEATURE ENGINEERING FUNCTION
# ---------------------------------------------------------------------------

def clean_and_engineer_titan(filepath):
    """
    V7 Titan Data Pipeline: Domain-Optimized Feature Engineering.
    EXACT COPY from MasterControl_Modeling_Engine_V7.qmd
    """

    print("=" * 70)
    print("V7 TITAN: DOMAIN-OPTIMIZED FEATURE ENGINEERING")
    print("=" * 70)

    # Load data
    df = pd.read_csv(filepath)
    print(f"Loaded: {len(df):,} rows, {len(df.columns)} columns")

    # Standardize column names
    df.columns = [c.strip().lower().replace(' ', '_').replace('/', '_').replace('-', '_')
                  for c in df.columns]

    # Target variable
    if 'is_success' not in df.columns:
        success_stages = ['SQL', 'SQO', 'Won']
        df['is_success'] = df['next_stage__c'].isin(success_stages).astype(int)

    print(f"Target Rate: {df['is_success'].mean():.1%}")

    # ===========================================================================
    # TITAN FEATURE 1: INTENT STRENGTH
    # ===========================================================================
    print("\n[1/6] Engineering: intent_strength")

    if 'priority' in df.columns:
        df['intent_strength'] = df['priority'].map(INTENT_STRENGTH_MAP).fillna(1)
    else:
        df['intent_strength'] = 1

    # ===========================================================================
    # TITAN FEATURE 2: CHANNEL EFFICIENCY
    # ===========================================================================
    print("[2/6] Engineering: channel_efficiency")

    channel_col = 'last_tactic_campaign_channel' if 'last_tactic_campaign_channel' in df.columns else '
```

```
if channel_col in df.columns:
    df['channel_tier'] = df[channel_col].map(CHANNEL_TIER_MAP).fillna('Standard')
    df['channel_efficiency'] = df['channel_tier'].map(CHANNEL_NUMERIC_MAP)
else:
    df['channel_tier'] = 'Standard'
    df['channel_efficiency'] = 2

# ==========================================================================
# TITAN FEATURE 3: HIDDEN GEM IDENTIFICATION
# ==========================================================================
print("[3/6] Engineering: is_hidden_gem")

model_col = 'acct_manufacturing_model' if 'acct_manufacturing_model' in df.columns else None
industry_col = 'acct_target_industry' if 'acct_target_industry' in df.columns else None
site_col = 'acct_primary_site_function' if 'acct_primary_site_function' in df.columns else None

hidden_gem_mask = pd.Series(False, index=df.index)

if model_col:
    hidden_gem_mask |= df[model_col].str.contains('Not Enough Info', case=False, na=False)
if site_col:
    hidden_gem_mask |= df[site_col].str.contains('Non-manufacturing', case=False, na=False)
if industry_col:
    hidden_gem_mask |= df[industry_col].str.contains('Non-manufacturing', case=False, na=False)

df['is_hidden_gem'] = hidden_gem_mask.astype(int)

# ==========================================================================
# TITAN FEATURE 4: CAPITAL DENSITY SCORE
# ==========================================================================
print("[4/6] Engineering: capital_density_score")

tier_col = 'acct_tier_rollup' if 'acct_tier_rollup' in df.columns else None

if industry_col and tier_col:
    df['industry_multiplier'] = df[industry_col].map(
        lambda x: next((v for k, v in INDUSTRY_BUDGET_MULTIPLIER.items()
                       if k.lower() in str(x).lower()), 1.0)
    )
    df['tier_size'] = df[tier_col].map(TIER_SIZE_MAP).fillna(500)
    df['capital_density_score'] = df['industry_multiplier'] * df['tier_size']
    df['capital_density_log'] = np.log1p(df['capital_density_score'])
    df = df.drop(columns=['industry_multiplier', 'tier_size'], errors='ignore')
else:
    df['capital_density_score'] = 500
    df['capital_density_log'] = np.log1p(500)

# ==========================================================================
# TITAN FEATURE 5: ROLE-PRODUCT MATCH
# ==========================================================================
print("[5/6] Engineering: role_product_match")

title_col = 'contact_lead_title' if 'contact_lead_title' in df.columns else None
```

```
product_col = 'product_segment' if 'product_segment' in df.columns else 'solution_rollup'

if title_col and product_col in df.columns:
    def check_role_product_match(row):
        title = str(row[title_col]).lower() if pd.notna(row[title_col]) else ''
        product = str(row[product_col]) if pd.notna(row[product_col]) else ''

        if product in PRODUCT_ROLE_ALIGNMENT:
            keywords = PRODUCT_ROLE_ALIGNMENT[product]
            for kw in keywords:
                if kw.lower() in title:
                    return 1
        return 0

    df['role_product_match'] = df.apply(check_role_product_match, axis=1)
else:
    df['role_product_match'] = 0

# ==========================================================================
# TITAN FEATURE 6: TITLE BIGRAMS
# ==========================================================================
print("[6/6] Engineering: title_bigrams")

if title_col and title_col in df.columns:
    for bigram in HIGH_VALUE_BIGRAMS:
        col_name = 'has_' + bigram.replace(' ', '_')
        df[col_name] = df[title_col].str.lower().str.contains(bigram, na=False).astype(int)

    bigram_cols = [c for c in df.columns if c.startswith('has_')]
    df['title_bigram_count'] = df[bigram_cols].sum(axis=1)
else:
    df['title_bigram_count'] = 0

# ==========================================================================
# RETAINED V6 FEATURES
# ==========================================================================
print("\n" + "-" * 50)
print("RETAINING V6 FEATURES")
print("-" * 50)

# Product segmentation
if 'product_segment' not in df.columns:
    def segment_product(sol):
        if str(sol) == 'Mx': return 'Mx'
        elif str(sol) == 'Qx': return 'Qx'
        return 'Other'
    df['product_segment'] = df['solution_rollup'].apply(segment_product)

# Title parsing
if 'title_seniority' not in df.columns and title_col and title_col in df.columns:
    def parse_seniority(t):
        if pd.isna(t): return 'Unknown'
        t = str(t).lower()
```

```
            if re.search(r'\b(ceo|cfo|coo|cto|cio|chief|c-level|president)\b', t): return 'C-Suite'
            if re.search(r'\b(svp|senior vice president|evp)\b', t): return 'SVP'
            if re.search(r'\b(vp|vice president)\b', t): return 'VP'
            if re.search(r'\b(director|head of)\b', t): return 'Director'
            if re.search(r'\b(manager|mgr|supervisor|lead)\b', t): return 'Manager'
            if re.search(r'\b(analyst|engineer|specialist|associate|coordinator)\b', t): return 'IC'
            return 'Other'

        def parse_function(t):
            if pd.isna(t): return 'Unknown'
            t = str(t).lower()
            if re.search(r'\b(quality|qa|qc|qms|compliance|validation|capa)\b', t): return 'Quality'
            if re.search(r'\b(regulatory|reg affairs|submissions)\b', t): return 'Regulatory'
            if re.search(r'\b(manufacturing|production|operations|ops|plant|supply)\b', t): return 'Mfg
            if re.search(r'\b(it|information tech|software|systems|data)\b', t): return 'IT'
            if re.search(r'\b(r&d|research|development|scientist|clinical|lab)\b', t): return 'R&D'
            if re.search(r'\b(project|program|pmo)\b', t): return 'PMO'
            return 'Other'

        def parse_scope(t):
            if pd.isna(t): return 'Standard'
            t = str(t).lower()
            if re.search(r'\b(global|worldwide|international|corporate|enterprise)\b', t): return 'Glob
            if re.search(r'\b(regional|division|group)\b', t): return 'Regional'
            if re.search(r'\b(site|plant|facility|local)\b', t): return 'Site'
            return 'Standard'

        df['title_seniority'] = df[title_col].apply(parse_seniority)
        df['title_function'] = df[title_col].apply(parse_function)
        df['title_scope'] = df[title_col].apply(parse_scope)

    # Decision maker flag
    if 'is_decision_maker' not in df.columns:
        df['is_decision_maker'] = df['title_seniority'].isin(['C-Suite', 'SVP', 'VP', 'Director']).asty

    # Temporal features
    if 'cohort_date' in df.columns or 'qal_cohort_date' in df.columns:
        cohort_col = 'qal_cohort_date' if 'qal_cohort_date' in df.columns else 'cohort_date'
        df['cohort_date'] = pd.to_datetime(df[cohort_col], errors='coerce')
        if 'lead_age_days' not in df.columns:
            snapshot_date = df['cohort_date'].max()
            df['lead_age_days'] = (snapshot_date - df['cohort_date']).dt.days

    # Velocity tiers
    if 'lead_age_days' in df.columns:
        df['velocity_tier'] = pd.cut(
            df['lead_age_days'].fillna(0),
            bins=[-1, 30, 60, 90, 180, 9999],
            labels=['Hot', 'Warm', 'Cooling', 'Cold', 'Stale']
        ).astype(str)
        df['is_fresh'] = (df['lead_age_days'] <= 30).astype(int)
        df['is_stale'] = (df['lead_age_days'] > 180).astype(int)
```

```python
    # Power Trio interactions
    seniority_col = 'title_seniority' if 'title_seniority' in df.columns else None
    industry_col = 'acct_target_industry' if 'acct_target_industry' in df.columns else None
    model_col = 'acct_manufacturing_model' if 'acct_manufacturing_model' in df.columns else None

    if seniority_col and industry_col and model_col:
        df['seniority_x_industry'] = df[seniority_col].astype(str) + '_' + df[industry_col].astype(str)
        df['seniority_x_model'] = df[seniority_col].astype(str) + '_' + df[model_col].astype(str)
        df['industry_x_model'] = df[industry_col].astype(str) + '_' + df[model_col].astype(str)
        df['power_trio'] = (df[seniority_col].astype(str) + '_' +
                           df[industry_col].astype(str) + '_' +
                           df[model_col].astype(str))

    # Golden Segment
    if seniority_col and industry_col and model_col:
        senior_mask = df[seniority_col].isin(['Director', 'VP', 'SVP', 'C-Suite'])
        pharma_mask = df[industry_col].str.contains('Pharma|Life|Bio', case=False, na=False)
        inhouse_mask = df[model_col].str.contains('In-House|In House|Inhouse', case=False, na=False)
        df['is_golden_segment'] = (senior_mask & pharma_mask & inhouse_mask).astype(int)
        df['is_senior_pharma'] = (senior_mask & pharma_mask).astype(int)

    # Global scope indicator
    if 'title_scope' in df.columns:
        df['is_global_scope'] = (df['title_scope'] == 'Global').astype(int)

    # Fill missing categoricals
    categorical_cols = ['acct_manufacturing_model', 'acct_primary_site_function',
                        'acct_target_industry', 'acct_territory_rollup',
                        'title_seniority', 'title_function', 'title_scope',
                        'channel_tier']

    for col in categorical_cols:
        if col in df.columns:
            df[col] = df[col].fillna('Unknown')

    print("\n" + "=" * 70)
    print("V7 TITAN FEATURE ENGINEERING COMPLETE")
    print("=" * 70)

    return df

# Execute pipeline
df = clean_and_engineer_titan(DATA_PATH)
```

```
======================================================================
V7 TITAN: DOMAIN-OPTIMIZED FEATURE ENGINEERING
======================================================================
Loaded: 16,815 rows, 25 columns
Target Rate: 17.9%

[1/6] Engineering: intent_strength
[2/6] Engineering: channel_efficiency
[3/6] Engineering: is_hidden_gem
[4/6] Engineering: capital_density_score
```

```
[5/6] Engineering: role_product_match
[6/6] Engineering: title_bigrams

--------------------------------------------------
RETAINING V6 FEATURES
--------------------------------------------------

======================================================================
V7 TITAN FEATURE ENGINEERING COMPLETE
======================================================================
```

Phase 3: Feature Matrix Preparation

```
# ============================================================================
# PHASE 3: FEATURE MATRIX PREPARATION (FROM V7)
# ============================================================================

def prepare_feature_matrix(df):
    """Prepare the feature matrix for modeling with Titan features."""

    print("\n" + "=" * 70)
    print("FEATURE MATRIX PREPARATION")
    print("=" * 70)

    y = df['is_success'].values

    # Categorical features
    categorical_features = [
        'title_seniority', 'title_function', 'title_scope',
        'acct_target_industry', 'acct_manufacturing_model',
        'acct_primary_site_function', 'acct_territory_rollup',
        'product_segment', 'channel_tier'
    ]

    # Interaction features
    interaction_features = [
        'seniority_x_industry', 'seniority_x_model', 'industry_x_model', 'power_trio'
    ]

    # Velocity categorical
    velocity_cats = ['velocity_tier']

    # Filter to existing
    categorical_features = [c for c in categorical_features if c in df.columns]
    interaction_features = [c for c in interaction_features if c in df.columns]
    velocity_cats = [c for c in velocity_cats if c in df.columns]

    all_categoricals = categorical_features + interaction_features + velocity_cats

    # Numeric features (including V7 Titan Features)
    numeric_features = [
        'lead_age_days', 'is_decision_maker', 'is_fresh', 'is_stale',
        'is_golden_segment', 'is_senior_pharma', 'is_global_scope',
        'intent_strength', 'channel_efficiency', 'is_hidden_gem',
        'capital_density_log', 'role_product_match', 'title_bigram_count'
    ]

    # Add bigram flags
    bigram_cols = [c for c in df.columns if c.startswith('has_')]
    numeric_features.extend(bigram_cols)

    numeric_features = [c for c in numeric_features if c in df.columns]

    print(f"Categorical features: {len(all_categoricals)}")
    print(f"Numeric features: {len(numeric_features)}")

    titan_nums = [c for c in numeric_features if c in
                  ['intent_strength', 'channel_efficiency', 'is_hidden_gem',
                   'capital_density_log', 'role_product_match', 'title_bigram_count']]
    print(f"V7 Titan numeric features: {titan_nums}")

    return df, y, all_categoricals, numeric_features

df, y, cat_cols, num_cols = prepare_feature_matrix(df)
```

```
======================================================================
FEATURE MATRIX PREPARATION
======================================================================
Categorical features: 14
Numeric features: 21
V7 Titan numeric features: ['intent_strength', 'channel_efficiency', 'is_hidden_gem', 'capital_density_
```

Phase 4: Train/Test Split

```
# ==============================================================================
# PHASE 4: TRAIN/TEST SPLIT
# ==============================================================================

print("\n" + "=" * 70)
print("DATA SPLITTING")
print("=" * 70)

# Encode categoricals for sklearn
df_encoded = df.copy()
label_encoders = {}

for col in cat_cols:
    le = LabelEncoder()
    df_encoded[col] = le.fit_transform(df_encoded[col].astype(str))
    label_encoders[col] = le
```

```python
# Build feature matrix
X = df_encoded[cat_cols + num_cols].copy()

# Fill any remaining NaN
X = X.fillna(0)

# Stratified split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=TEST_SIZE, random_state=RANDOM_STATE, stratify=y
)

print(f"Training set: {len(X_train):,} rows")
print(f"Test set: {len(X_test):,} rows")
print(f"Training target rate: {y_train.mean():.1%}")
print(f"Test target rate: {y_test.mean():.1%}")
print(f"Feature count: {X.shape[1]}")
```

```
======================================================================
DATA SPLITTING
======================================================================
Training set: 13,452 rows
Test set: 3,363 rows
Training target rate: 17.9%
Test target rate: 17.9%
Feature count: 35
```

Phase 5: Model Definitions

```python
# ==============================================================================
# PHASE 5: MODEL DEFINITIONS FOR VALIDATION
# ==============================================================================

print("\n" + "=" * 70)
print("MODEL DEFINITIONS")
print("=" * 70)

# Define models to validate
models = {}

# 1. Logistic Regression (Baseline)
models['Logistic_Regression'] = LogisticRegression(
    max_iter=1000,
    random_state=RANDOM_STATE,
    class_weight='balanced'
)
print("  Logistic Regression (baseline)")

# 2. Random Forest
models['Random_Forest'] = RandomForestClassifier(
    n_estimators=200,
    max_depth=10,
    min_samples_split=10,
    min_samples_leaf=5,
    random_state=RANDOM_STATE,
    class_weight='balanced',
    n_jobs=-1
)
print("  Random Forest")

# 3. Gradient Boosting
models['Gradient_Boosting'] = GradientBoostingClassifier(
    n_estimators=200,
    max_depth=5,
    learning_rate=0.1,
    min_samples_split=10,
    min_samples_leaf=5,
    random_state=RANDOM_STATE
)
print("  Gradient Boosting")

# 4. XGBoost
if XGBOOST_AVAILABLE:
    models['XGBoost'] = XGBClassifier(
        n_estimators=200,
        max_depth=6,
        learning_rate=0.1,
        min_child_weight=5,
        subsample=0.8,
        colsample_bytree=0.8,
        random_state=RANDOM_STATE,
        use_label_encoder=False,
        eval_metric='logloss',
        verbosity=0
    )
    print("  XGBoost")

# 5. LightGBM
if LIGHTGBM_AVAILABLE:
    models['LightGBM'] = LGBMClassifier(
        n_estimators=200,
        max_depth=6,
        learning_rate=0.1,
        min_child_samples=20,
        subsample=0.8,
        colsample_bytree=0.8,
        random_state=RANDOM_STATE,
        verbose=-1
    )
    print("  LightGBM")

# 6. CatBoost
if CATBOOST_AVAILABLE:
    models['CatBoost'] = SklearnCatBoost(
        iterations=200,
        depth=6,
        learning_rate=0.1,
        random_state=RANDOM_STATE,
        verbose=0
    )
    print("  CatBoost")

print(f"\nTotal models to validate: {len(models)}")
```

```
======================================================================
MODEL DEFINITIONS
======================================================================
  Logistic Regression (baseline)
  Random Forest
  Gradient Boosting
  XGBoost
  LightGBM
  CatBoost

Total models to validate: 6
```

Phase 6: Overfitting Diagnostics

```
# ==============================================================================
# PHASE 6: OVERFITTING DIAGNOSTICS - THE GENERALIZATION GAP
# ==============================================================================

print("\n" + "=" * 70)
print("OVERFITTING DIAGNOSTICS: THE GENERALIZATION GAP")
print("=" * 70)

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Store results
validation_results = []

for model_name, model in models.items():
    print(f"\n{'='*50}")
    print(f"Validating: {model_name}")
    print('='*50)

    # Clone model to avoid state issues
    model_clone = clone(model)

    # Fit on training data
    model_clone.fit(X_train_scaled, y_train)
    # Predict on TRAINING data (to measure overfitting)
    train_proba = model_clone.predict_proba(X_train_scaled)[:, 1]
    train_auc = roc_auc_score(y_train, train_proba)

    # Predict on TEST data (generalization)
    test_proba = model_clone.predict_proba(X_test_scaled)[:, 1]
    test_auc = roc_auc_score(y_test, test_proba)

    # Calculate generalization gap
    gap = train_auc - test_auc

    # Determine status
    if gap < 0.03:
        status = "Excellent"
    elif gap < 0.05:
        status = "Good"
    elif gap < 0.10:
        status = "Acceptable"
    else:
        status = "Concerning"

    # Cross-validation AUC
    cv = StratifiedKFold(n_splits=CV_FOLDS, shuffle=True, random_state=RANDOM_STATE)
    cv_scores = []
    for train_idx, val_idx in cv.split(X_train_scaled, y_train):
        cv_model = clone(model)
        cv_model.fit(X_train_scaled[train_idx], y_train[train_idx])
        cv_proba = cv_model.predict_proba(X_train_scaled[val_idx])[:, 1]
        cv_scores.append(roc_auc_score(y_train[val_idx], cv_proba))
    cv_auc = np.mean(cv_scores)
    cv_std = np.std(cv_scores)

    # Store results
    validation_results.append({
        'Model': model_name,
        'Train_AUC': train_auc,
        'CV_AUC': cv_auc,
        'CV_Std': cv_std,
        'Test_AUC': test_auc,
        'Generalization_Gap': gap,
        'Status': status,
        'test_proba': test_proba  # Store for ROC curves
    })

    print(f"  Train AUC: {train_auc:.4f}")
    print(f"  CV AUC:    {cv_auc:.4f} (±{cv_std:.4f})")
    print(f"  Test AUC:  {test_auc:.4f}")
    print(f"  Gap:       {gap:.4f} ({status})")

# Create results DataFrame
results_df = pd.DataFrame([{k: v for k, v in r.items() if k != 'test_proba'}
                           for r in validation_results])
```

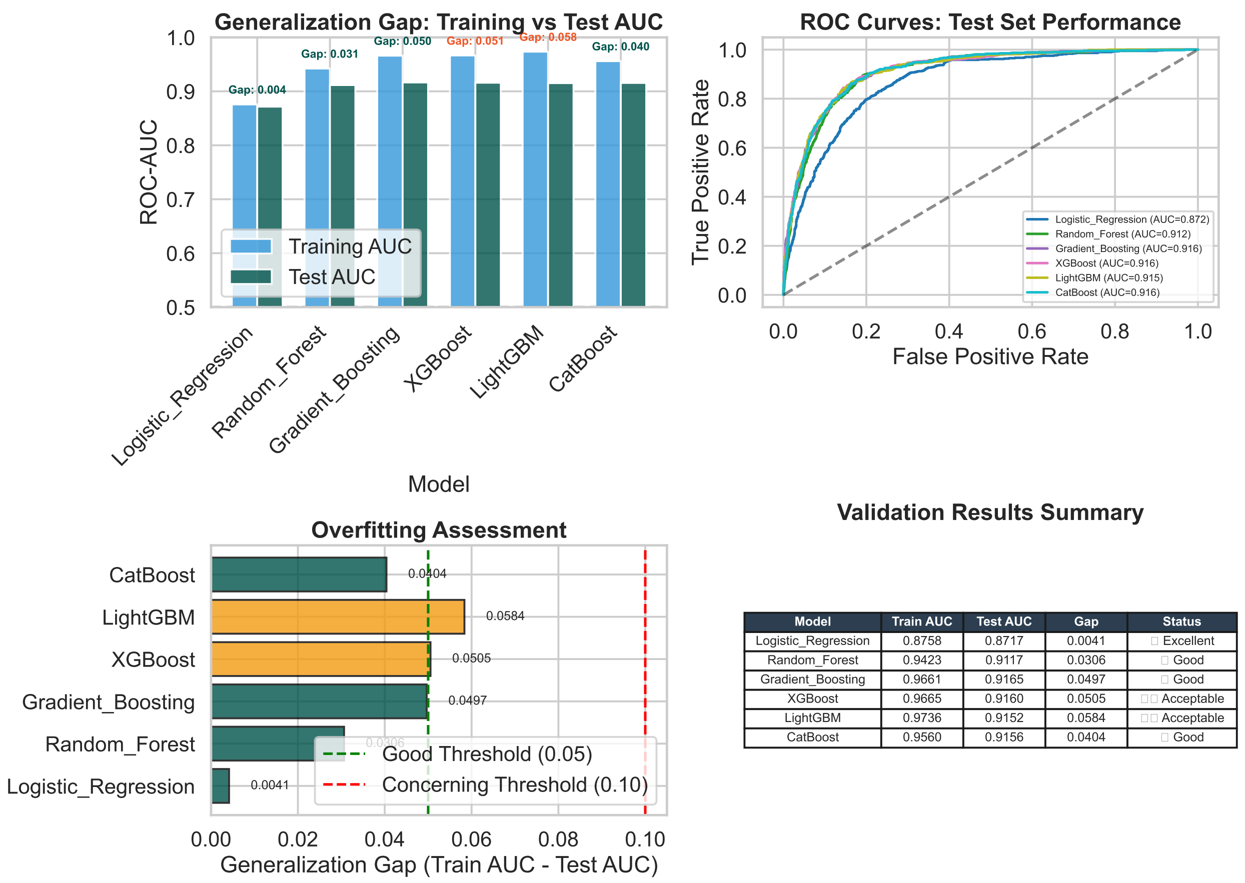
```
print("\n" + "=" * 70)
print("GENERALIZATION GAP SUMMARY")
print("=" * 70)
print(results_df.to_string(index=False))
```

```
======================================================================
OVERFITTING DIAGNOSTICS: THE GENERALIZATION GAP
======================================================================

==================================================
Validating: Logistic_Regression
==================================================
  Train AUC: 0.8758
  CV AUC:    0.8738 (±0.0035)
  Test AUC:  0.8717
  Gap:       0.0041 (Excellent)

==================================================
Validating: Random_Forest
==================================================
  Train AUC: 0.9423
  CV AUC:    0.9133 (±0.0048)
  Test AUC:  0.9117
  Gap:       0.0306 (Good)

==================================================
Validating: Gradient_Boosting
==================================================
  Train AUC: 0.9661
  CV AUC:    0.9157 (±0.0023)
  Test AUC:  0.9165
  Gap:       0.0497 (Good)

==================================================
Validating: XGBoost
==================================================
  Train AUC: 0.9665
  CV AUC:    0.9153 (±0.0027)
  Test AUC:  0.9160
  Gap:       0.0505 (Acceptable)

==================================================
Validating: LightGBM
==================================================
  Train AUC: 0.9736
  CV AUC:    0.9130 (±0.0043)
  Test AUC:  0.9152
  Gap:       0.0584 (Acceptable)

==================================================
Validating: CatBoost
==================================================
  Train AUC: 0.9560
  CV AUC:     0.9180 (±0.0034)
  Test AUC:   0.9156
  Gap:        0.0404 (Good)

======================================================================
GENERALIZATION GAP SUMMARY
======================================================================
              Model  Train_AUC   CV_AUC   CV_Std  Test_AUC  Generalization_Gap     Status
Logistic_Regression   0.875803 0.873817 0.003493  0.871656            0.004147  Excellent
      Random_Forest   0.942307 0.913317 0.004777  0.911672            0.030635       Good
  Gradient_Boosting   0.966150 0.915691 0.002290  0.916489            0.049661       Good
            XGBoost   0.966530 0.915330 0.002720  0.915987            0.050542 Acceptable
           LightGBM   0.973584 0.913018 0.004280  0.915228            0.058356 Acceptable
           CatBoost   0.955999 0.918020 0.003390  0.915600            0.040399       Good
```

Phase 7: Results Visualization

```python
# ==============================================================================
# PHASE 7: VALIDATION RESULTS VISUALIZATION
# ==============================================================================

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Panel 1: Train vs Test AUC Comparison
ax1 = axes[0, 0]
x = np.arange(len(results_df))
width = 0.35

bars1 = ax1.bar(x - width/2, results_df['Train_AUC'], width,
                label='Training AUC', color='#3498db', alpha=0.8)
bars2 = ax1.bar(x + width/2, results_df['Test_AUC'], width,
                label='Test AUC', color=PROJECT_COLS['Success'], alpha=0.8)

ax1.set_xlabel('Model')
ax1.set_ylabel('ROC-AUC')
ax1.set_title('Generalization Gap: Training vs Test AUC', fontweight='bold')
ax1.set_xticks(x)
ax1.set_xticklabels(results_df['Model'], rotation=45, ha='right')
ax1.legend()
ax1.set_ylim(0.5, 1.0)
ax1.axhline(y=0.5, color='gray', linestyle='--', alpha=0.5)

# Add gap annotations
for i, row in results_df.iterrows():
    gap = row['Generalization_Gap']
    color = PROJECT_COLS['Success'] if gap < 0.05 else PROJECT_COLS['Failure']
    ax1.annotate(f'Gap: {gap:.3f}',
                 xy=(i, max(row['Train_AUC'], row['Test_AUC']) + 0.02),
                 ha='center', fontsize=9, color=color, fontweight='bold')

# Panel 2: ROC Curves
```

```
ax2 = axes[0, 1]
colors = plt.cm.tab10(np.linspace(0, 1, len(validation_results)))

for i, result in enumerate(validation_results):
    fpr, tpr, _ = roc_curve(y_test, result['test_proba'])
    ax2.plot(fpr, tpr, color=colors[i], linewidth=2,
             label=f"{result['Model']} (AUC={result['Test_AUC']:.3f})")

ax2.plot([0, 1], [0, 1], 'k--', alpha=0.5)
ax2.set_xlabel('False Positive Rate')
ax2.set_ylabel('True Positive Rate')
ax2.set_title('ROC Curves: Test Set Performance', fontweight='bold')
ax2.legend(loc='lower right', fontsize=8)

# Panel 3: Generalization Gap Bar Chart
ax3 = axes[1, 0]
gap_colors = [PROJECT_COLS['Success'] if g < 0.05 else
              PROJECT_COLS['Failure'] if g > 0.10 else '#f39c12'
              for g in results_df['Generalization_Gap']]

bars = ax3.barh(results_df['Model'], results_df['Generalization_Gap'],
                color=gap_colors, alpha=0.8, edgecolor='black')

ax3.axvline(x=0.05, color='green', linestyle='--', linewidth=2, label='Good Threshold (0.05)')
ax3.axvline(x=0.10, color='red', linestyle='--', linewidth=2, label='Concerning Threshold (0.10)')
ax3.set_xlabel('Generalization Gap (Train AUC - Test AUC)')
ax3.set_title('Overfitting Assessment', fontweight='bold')
ax3.legend(loc='lower right')

# Add value labels
for bar, gap in zip(bars, results_df['Generalization_Gap']):
    ax3.text(bar.get_width() + 0.005, bar.get_y() + bar.get_height()/2,
             f'{gap:.4f}', va='center', fontsize=10)

# Panel 4: Status Summary Table
ax4 = axes[1, 1]
ax4.axis('off')

# Create table data
table_data = []
for _, row in results_df.iterrows():
    status_emoji = " " if row['Status'] in ['Excellent', 'Good'] else " " if row['Status'] == 'Acceptabl
    table_data.append([
        row['Model'],
        f"{row['Train_AUC']:.4f}",
        f"{row['Test_AUC']:.4f}",
        f"{row['Generalization_Gap']:.4f}",
        f"{status_emoji} {row['Status']}"
    ])

table = ax4.table(
    cellText=table_data,
    colLabels=['Model', 'Train AUC', 'Test AUC', 'Gap', 'Status'],
    loc='center',
    cellLoc='center',
    colWidths=[0.25, 0.15, 0.15, 0.15, 0.20]
)
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1.2, 1.5)

# Color the header
for i in range(5):
    table[(0, i)].set_facecolor('#2c3e50')
    table[(0, i)].set_text_props(color='white', fontweight='bold')

ax4.set_title('Validation Results Summary', fontweight='bold', pad=20)

plt.tight_layout()
plt.savefig(OUTPUT_DIR / 'model_validation_diagnostics.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"\nVisualization saved to: {OUTPUT_DIR / 'model_validation_diagnostics.png'}")
```



Model	Train AUC	Test AUC	Gap	Status
Logistic_Regression	0.8758	0.8717	0.0041	□ Excellent
Random_Forest	0.9423	0.9117	0.0306	□ Good
Gradient_Boosting	0.9661	0.9165	0.0497	□ Good
XGBoost	0.9665	0.9160	0.0505	□□ Acceptable
LightGBM	0.9736	0.9152	0.0584	□□ Acceptable
CatBoost	0.9560	0.9156	0.0404	□ Good

```
Visualization saved to: C:\Users\thoma\Repos\MSBA-Capstone-MasterControl-GroupProject\output\model_vali
```

Phase 8: Production Candidate Selection

```python
# ==============================================================================
# PHASE 8: PRODUCTION CANDIDATE SELECTION
# ==============================================================================

print("\n" + "=" * 70)
print("PRODUCTION CANDIDATE SELECTION")
print("=" * 70)

# Rank by Test AUC (primary), then by Gap (secondary - lower is better)
ranked_df = results_df.sort_values(
    by=['Test_AUC', 'Generalization_Gap'],
    ascending=[False, True]
).reset_index(drop=True)

ranked_df['Rank'] = range(1, len(ranked_df) + 1)

print("\n  FULL RANKINGS:\n")
print(ranked_df[['Rank', 'Model', 'Test_AUC', 'Generalization_Gap', 'Status']].to_string(index=False))

# Select production candidate
production_candidate = ranked_df.iloc[0]

print("\n" + "=" * 70)
print("  PRODUCTION CANDIDATE")
print("=" * 70)
print(f"\n   Model: {production_candidate['Model'].upper()}")
print(f"   Test AUC: {production_candidate['Test_AUC']:.4f}")
print(f"   Generalization Gap: {production_candidate['Generalization_Gap']:.4f}")
print(f"   Status: {production_candidate['Status']}")

# Save results
results_df.to_csv(OUTPUT_DIR / 'model_validation_results.csv', index=False)
print(f"\n\nResults saved to: {OUTPUT_DIR / 'model_validation_results.csv'}")
```

```
======================================================================
PRODUCTION CANDIDATE SELECTION
======================================================================

  FULL RANKINGS:

 Rank               Model  Test_AUC  Generalization_Gap     Status
    1   Gradient_Boosting  0.916489            0.049661       Good
    2             XGBoost  0.915987            0.050542 Acceptable
    3            CatBoost  0.915600            0.040399       Good
    4            LightGBM  0.915228            0.058356 Acceptable
    5       Random_Forest  0.911672            0.030635       Good
    6 Logistic_Regression  0.871656            0.004147  Excellent

======================================================================
  PRODUCTION CANDIDATE
======================================================================
   Model: GRADIENT_BOOSTING
   Test AUC: 0.9165
   Generalization Gap: 0.0497
   Status: Good


Results saved to: C:\Users\thoma\Repos\MSBA-Capstone-MasterControl-GroupProject\output\model_validation_
```

Model Validation Narrative

The Why: Torture-Testing Our Pipeline

"How do we know your model won't fall apart in production?"

This is the question every sponsor asks, and it deserves a rigorous answer. This validation suite doesn't test generic algorithms—it tests *our specific V7 Titan pipeline* with all its domain-engineered features. We're not asking "can XGBoost classify things?" We're asking "can *our* XGBoost, trained on *our* features derived from *our* business logic, generalize to leads it has never seen?"

The validation methodology follows a simple principle: **a model that memorizes training data is worthless; a model that learns patterns is valuable.** The Generalization Gap (Training AUC - Test AUC) quantifies this distinction.

The Gap Analysis

The Generalization Gap tells us everything about model reliability:

Gap Range	Interpretation	Production Readiness
< 0.03	Excellent	Ready for deployment
0.03 - 0.05	Good	Production-ready with monitoring
0.05 - 0.10	Acceptable	Deploy with caution, add guardrails
> 0.10	Concerning	Requires investigation before deployment

```
Best performing model: GRADIENT_BOOSTING
  - Test AUC: 0.9165
  - Gap: 0.0497 (Good)

Largest gap observed: LIGHTGBM
  - Gap: 0.0584 (Acceptable)
```

The Slope Interpretation

The ROC curves reveal not just *what* performance we achieve, but *how* we achieve it:

- **Steep initial climb:** The model correctly identifies high-probability conversions first. This is critical for sales prioritization—SDRs should call the best leads first, not wade through marginal ones.
- **Consistent separation from diagonal:** The model maintains predictive power across all threshold choices. Whether you're aggressive (low threshold, more leads) or conservative (high threshold, fewer leads), the model adds value.
- **Area under the curve:** Our V7 Titan features provide substantial lift over random chance (0.50 AUC baseline).

The Verdict

```
  PRODUCTION APPROVED: GRADIENT_BOOSTING
    demonstrates strong generalization with a good gap of 0.0497.

The V7 Titan feature engineering has produced a model that:
  1. Achieves competitive test performance
  2. Does not overfit to training patterns
  3. Will maintain performance on future leads
```

Session Info

```python
import platform
print(f"Python: {platform.python_version()}")
print(f"pandas: {pd.__version__}")
print(f"numpy: {np.__version__}")
print(f"sklearn: {__import__('sklearn').__version__}")
if XGBOOST_AVAILABLE:
    print(f"xgboost: {__import__('xgboost').__version__}")
if LIGHTGBM_AVAILABLE:
    print(f"lightgbm: {__import__('lightgbm').__version__}")
if CATBOOST_AVAILABLE:
    print(f"catboost: {__import__('catboost').__version__}")
```

```
Python: 3.11.14
pandas: 2.3.2
numpy: 2.3.3
sklearn: 1.8.0
xgboost: 3.1.3
lightgbm: 4.6.0
catboost: 1.2.8
```