Model Training Pipeline Documentation

Overview

This document provides a comprehensive guide to the machine learning model training pipeline for predicting the next best product recommendation for wealth management clients. The pipeline processes historical client data to train a LightGBM multiclass classification model.

Table of Contents

---

1. Data Source and Identification

1.1 Source Table

- **Table Name**: `dl_tenants_daas.us_wealth_management.wealth_management_client_metrics`
- **Location**: Unity Catalog (Databricks)
- **Description**: Contains comprehensive client metrics including policy information, financial data, demographics, and asset allocations

1.2 Initial Data Loading

```
df_raw = spark.table("dl_tenants_daas.us_wealth_management.wealth_management_client_metrics")
```

1.3 Data Deduplication

- **Purpose**: Remove duplicate policy records per client
- **Method**: `df_raw.dropDuplicates(["cont_id", "policy_no"])`
- **Rationale**: Ensures each policy is counted only once per client (`cont_id`)

---

2. Data Preparation and Filtering

2.1 Product Category Creation

Purpose: Map raw product fields to standardized product categories

Source Fields:

- `prod_lob` : Product line of business
- `sub_product_level_1` : First-level product subcategory
- `sub_product_level_2` : Second-level product subcategory

Product Categories (7 classes):

1. **LIFE_INSURANCE**: Life insurance products (VLI, WL, UL/IUL, TERM, etc.)
2. **RETIREMENT**: Retirement products (401K, IRA, 403B, SEP, etc.)
3. **INVESTMENT**: Investment products (Brokerage, Advisory, Direct)
4. **NETWORK_PRODUCTS**: Network-related products
5. **DISABILITY**: Disability insurance products
6. **HEALTH**: Health insurance products
7. **OTHER**: All other products

Mapping Logic: Hierarchical conditional logic based on `prod_lob` , `sub_product_level_1` , and `sub_product_level_2` fields.

2.2 Event Data Selection

Selected Columns:

- Product information: `product_category`

- Temporal data: `register_date`, `isrd_brth_date`
- Financial metrics: `acct_val_amt`, `face_amt`, `cash_val_amt`, `wc_total_assets`
- Asset mix: `wc_assetmix_stocks`, `wc_assetmix_bonds`, `wc_assetmix_mutual_funds`, `wc_assetmix_annuity`, `wc_assetmix_deposits`, `wc_assetmix_other_assets`
- Client identifiers: `cont_id`
- Demographics: `psn_age`, `client_seg`, `client_seg_1`, `aum_band`
- Channel information: `channel`, `agent_segment`, `branchoffice_code`
- Status: `policy_status`

2.3 Data Quality Filters

Applied Filters:

```
.filter(
    (F.col("cont_id").isNotNull()) &
    (F.col("register_date").isNotNull()) &
    (F.col("product_category").isNotNull()) &
    (F.col("policy_status") == "Active")
)
```

Rationale:

- Only **Active** policies are considered (excludes cancelled/terminated policies)
- Removes records with missing critical fields
- Ensures data quality for temporal analysis

2.4 Data Sampling (Optional)

- **Parameter**: `SAMPLE_FRACTION = 0.6` (60% of data)
- **Method**: `df_events.sample(withReplacement=False, fraction=0.6, seed=42)`

2.5 Temporal Ordering

Purpose: Order policies chronologically per client to identify first and second policies

Steps:

1. Convert dates to timestamps:
    - `register_ts = F.to_timestamp("register_date")`
    - `birth_ts = F.to_timestamp("isrd_brth_date")`
2. Create event index using window function:

    ```
    w = Window.partitionBy("cont_id").orderBy("register_ts")
    df_events = df_events.withColumn("event_idx", F.row_number().over(w))
    ```

    - `event_idx = 1`: First policy (earliest registration date)
    - `event_idx = 2`: Second policy

2.6 Client Filtering: Multi-Policy Clients Only

Critical Filter: Only clients with **2 or more policies** are used for training

Rationale:

- The model predicts the **second product** based on the **first product** and client characteristics

Implementation:

```
w_count = Window.partitionBy("cont_id")
df_events = df_events.withColumn("total_policies", F.count("*").over(w_count))
df_events_multi = df_events.filter(F.col("total_policies") >= 2)
```

Result:

- Training data contains only clients who have purchased at least 2 products
- Each training example = (first product features, second product label)

---

3. Feature Engineering

3.1 First and Second Policy Separation

First Policy Features (`df_first`)

Extracted from `event_idx == 1`:

- `first_product_category`: Product category of first policy
- `first_register_ts`: Registration timestamp of first policy
- `first_acct_val_amt`: Account value at first policy
- `first_face_amt`: Face amount of first policy
- `first_cash_val_amt`: Cash value of first policy
- Wealth metrics: `wc_total_assets`, `wc_assetmix_*` (at time of first policy)
- Demographics: `psn_age`, `client_seg`, `client_seg_1`, `aum_band`
- Channel: `channel`, `agent_segment`, `branchoffice_code`

Second Policy Target (`df_second`)

Extracted from `event_idx == 2`:

- `second_product_category`: **TARGET VARIABLE** (what we're predicting)
- `second_register_ts`: Registration timestamp of second policy

Join Operation

```
df_combined = df_first.join(df_second, on="cont_id", how="inner")
```

- **Join Type**: Inner join ensures we only keep clients with both first AND second policies
- **Result**: One row per client with first policy features and second policy label

3.2 New Feature Creation

A. Asset Allocation Ratios

Purpose: Normalize asset mix values by total assets to create proportional features

Features Created:

1. `stock_allocation_ratio = wc_assetmix_stocks / wc_total_assets`
2. `bond_allocation_ratio = wc_assetmix_bonds / wc_total_assets`
3. `annuity_allocation_ratio = wc_assetmix_annuity / wc_total_assets`
4. `mutual_fund_allocation_ratio = wc_assetmix_mutual_funds / wc_total_assets`

Handling Division by Zero: Returns `None` if `wc_total_assets == 0`, later imputed to 0

B. Temporal Features

Purpose: Capture timing and lifecycle information

1. **`season_of_first_policy`**: Quarter when first policy was purchased
    - Q1: Jan-Mar
    - Q2: Apr-Jun
    - Q3: Jul-Sep
    - Q4: Oct-Dec
2. **`age_at_first_policy`**: Client age when first policy was purchased
    - Formula: `datediff(first_register_ts, birth_ts) / 365.25`
    - Units: Years (decimal)
3. **`years_to_second_policy`**: Time elapsed between first and second policy
    - Formula: `datediff(second_register_ts, first_register_ts) / 365.25`
    - Units: Years (decimal)
    - **Key Feature**: Indicates client engagement speed

3.3 Target Variable Creation

Product Vocabulary Building

```
prod_list = df_combined.select("second_product_category").distinct().rdd.map(lambda r: r[0]).collect()
prod_list = sorted([p for p in prod_list if p is not None])
prod2id = {p: i for i, p in enumerate(prod_list)}   # 0-indexed
id2prod = {v: k for k, v in prod2id.items()}
NUM_CLASSES = len(prod2id)
```

Result:

- `prod2id`: Maps product category name → integer ID (0 to N-1)
- `id2prod`: Maps integer ID → product category name
- `NUM_CLASSES`: Number of unique product categories (typically 7)

Label Creation

```
df_combined = df_combined.withColumn(
    "label",
    F.udf(lambda x: prod2id.get(x, 0), IntegerType())(F.col("second_product_category"))
)
```

Label Range: 0 to (NUM_CLASSES - 1)

- Each integer represents a product category
- Used as target for multiclass classification

3.4 Missing Value Imputation

Categorical Features

Strategy: Fill with mode (most frequent value) **Features**:

- `first_product_category`, `client_seg`, `client_seg_1`, `aum_band`
- `channel`, `agent_segment`, `branchoffice_code`
- `season_of_first_policy`

Fallback: If mode calculation fails, use "UNKNOWN"

3.5 Categorical Feature Encoding

Purpose: Convert categorical strings to integer indices for LightGBM

Method: String-to-index mapping

```
for c in categorical_cols:
    vals = [r[0] for r in df_combined.select(c).distinct().collect()]
    m = {str(v): i for i, v in enumerate(sorted([str(x) for x in vals]))}
    # Broadcast mapping for efficient UDF execution
    b = spark.sparkContext.broadcast(m)
    df_combined = df_combined.withColumn(
        c + "_idx",
        F.udf(lambda s: int(b.value.get(str(s), 0)), IntegerType())(
            F.coalesce(F.col(c).cast("string"), F.lit("UNKNOWN"))
        )
    )
```

Result: Each categorical feature gets a corresponding `_idx` column

- `first_product_category` → `first_product_category_idx`
- `client_seg` → `client_seg_idx`
- etc.

Note: Original categorical columns are kept for reference but not used in model training

3.6 Final Feature Set

Total Features: 25 features

Numeric Features (13):

1. `first_acct_val_amt`
2. `first_face_amt`
3. `first_cash_val_amt`
4. `wc_total_assets`
5. `wc_assetmix_stocks`
6. `wc_assetmix_bonds`
7. `wc_assetmix_mutual_funds`
8. `wc_assetmix_annuity`
9. `wc_assetmix_deposits`
10. `wc_assetmix_other_assets`
11. `psn_age`
12. `age_at_first_policy`

13. `years_to_second_policy`

Allocation Ratio Features (4):

14. `stock_allocation_ratio`
15. `bond_allocation_ratio`
16. `annuity_allocation_ratio`
17. `mutual_fund_allocation_ratio`

Encoded Categorical Features (8):

18. `first_product_category_idx`
19. `client_seg_idx`
20. `client_seg_1_idx`
21. `aum_band_idx`
22. `channel_idx`
23. `agent_segment_idx`
24. `branchoffice_code_idx`
25. `season_of_first_policy_idx`

---

4. Train/Validation/Test Split

4.1 Split Methodology

Method: Random split using Spark's `randomSplit()`

Split Ratios:

- **Training**: 80% (`TRAIN_FRAC = 0.8`)
- **Validation**: 10% (`VAL_FRAC = 0.1`)
- **Test**: 10% (`TEST_FRAC = 0.1`)

Random Seed: `RANDOM_SEED = 42` (for reproducibility)

Implementation:

```
train_spark, val_spark, test_spark = df_combined.randomSplit(
    [TRAIN_FRAC, VAL_FRAC, TEST_FRAC],
    seed=RANDOM_SEED
)
```

4.2 Data Caching

Purpose: Speed up conversion to Pandas DataFrames

```
train_spark = train_spark.cache()
val_spark = val_spark.cache()
test_spark = test_spark.cache()
```

4.3 Conversion to Pandas

Purpose: LightGBM requires Pandas DataFrames for training

Selected Columns:

- `cont_id` : Client identifier (for tracking)
- `label` : Target variable
- All 25 feature columns (from `model_feature_cols`)

Final Cleanup:

```
train_pd.fillna(0, inplace=True)
val_pd.fillna(0, inplace=True)
test_pd.fillna(0, inplace=True)
```

4.4 Data Summary

Typical Results (with 60% sampling):

- **Training**: ~213,597 records
- **Validation**: ~26,451 records
- **Test**: ~27,392 records
- **Total**: ~267,440 records
- **Feature Count**: 25 features
- **Classes**: 7 product categories

---

5. Model Training

5.1 Model Architecture

Algorithm: LightGBM (Gradient Boosting Decision Tree) **Task**: Multiclass Classification **Objective**: Predict second product category (7 classes)

5.2 Initial Hyperparameters

```
LGB_PARAMS = {
    "objective": "multiclass",
    "num_class": NUM_CLASSES,  # Set dynamically (typically 7)
    "metric": "multi_logloss",
    "boosting_type": "gbdt",
    "learning_rate": 0.05,
    "num_leaves": 64,
    "min_data_in_leaf": 50,
    "feature_fraction": 0.8,
    "subsample": 0.8,
    "subsample_freq": 1,
    "lambda_l2": 2.0,
    "verbosity": -1
}
NUM_BOOST_ROUND = 2000
EARLY_STOP = 50
```

5.3 Training Process

```
train_ds = lgb.Dataset(train_pd[feature_cols_final], label=train_pd["label"])
val_ds = lgb.Dataset(val_pd[feature_cols_final], label=val_pd["label"], reference=train_ds)

model = lgb.train(
    LGB_PARAMS,
    train_ds,
    valid_sets=[train_ds, val_ds],
    valid_names=["train", "val"],
    num_boost_round=NUM_BOOST_ROUND,
    callbacks=[lgb.early_stopping(50)]
)
```

Key Components:

- **Early Stopping**: Stops training if validation loss doesn't improve for 50 rounds
- **Validation Monitoring**: Tracks both training and validation loss
- **Reference Dataset**: Validation dataset references training dataset for categorical feature consistency

5.4 Model Evaluation

Metrics Calculated:

1. **Accuracy**: Overall classification accuracy
2. **F1 Score (Weighted)**: Class-weighted F1 score

---

6. Hyperparameter Tuning

6.1 Tuning Framework

Tool: Optuna (Bayesian optimization) **Objective Metric**: F1-weighted score on validation set **Trials**: 35 trials (configurable)

6.2 Hyperparameter Search Space

```
params = {
    "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.1, log=True),
    "num_leaves": trial.suggest_int("num_leaves", 31, 127),
    "min_data_in_leaf": trial.suggest_int("min_data_in_leaf", 20, 100),
    "feature_fraction": trial.suggest_float("feature_fraction", 0.6, 1.0),
    "subsample": trial.suggest_float("subsample", 0.6, 1.0),
    "lambda_l2": trial.suggest_float("lambda_l2", 0.1, 10.0, log=True),
}
```

6.3 Optimization Process

1. **Sampler**: TPE (Tree-structured Parzen Estimator) with seed=42
2. **Direction**: Maximize F1-weighted score
3. **Early Stopping**: 50 rounds per trial
4. **MLflow Integration**: Logs all trials and best parameters

6.4 Best Hyperparameters

```
learning_rate: 0.0696
num_leaves: 127
min_data_in_leaf: 60
feature_fraction: 0.630
subsample: 0.762
lambda_l2: 0.328
```

6.5 Final Model Training

Process:

1. Train new model with best hyperparameters
2. Use same train/validation split
3. Evaluate on test set (unseen during tuning)

---

7. Model Registration

7.1 MLflow Model Registration

Model Name: `eda_smartlist.models.lgbm_model_hyperparameter_310126` **Version**: 1 **Location**: Unity Catalog

Registration Code:

```
with mlflow.start_run():
    mlflow.lightgbm.log_model(
        best_model,
        artifact_path="lgbm_model_hyperparameter_310126",
        registered_model_name="eda_smartlist.models.lgbm_model_hyperparameter_310126",
        signature=signature,
        input_example=input_example
    )
```

7.2 Model Signature

Purpose: Defines input/output schema for model serving **Input**: 25 feature columns (as defined in `model_feature_cols`) **Output**: Probability distribution over 7 classes

7.3 Model Loading

```
model = mlflow.lightgbm.load_model(
    "models:/eda_smartlist.models.lgbm_model_hyperparameter_310126/1"
)
```

---

8. Artifacts and Dependencies

8.1 Required Artifacts

The following artifacts must be saved and loaded for prediction:

1. **`prod2id`** : Dictionary mapping product category → integer ID
2. **`id2prod`** : Dictionary mapping integer ID → product category
3. **`label_map`** : Label mapping (for compatibility)
4. **`num_classes`** : Number of product classes (typically 7)
5. **`categorical_mappings`** : Dictionary of categorical feature encodings
    - Maps each categorical feature to its string→index mapping
    - Example: `{"first_product_category": {"LIFE_INSURANCE": 0, "RETIREMENT": 1, ...}}`
6. **`feature_cols`** : List of feature column names in correct order (25 features)

8.2 Artifact Storage

Location: Saved in `artifacts.pkl` file **Usage**: Loaded by `PreprocessAndPredictModel.load_context()`

8.3 Python Dependencies

- **PySpark**: Data processing and Spark SQL operations
- **LightGBM**: Model training and inference
- **Pandas**: Data manipulation
- **NumPy**: Numerical operations
- **MLflow**: Model tracking and registration
- **Optuna**: Hyperparameter optimization (optional)
- **SHAP**: Model interpretability (for talking points generation)

---

9. Connection to Prediction Pipeline

9.1 Prediction Pipeline Overview

The `final_pipeline.py` script uses the trained model to make predictions for **single-policy clients**.

9.2 Key Differences: Training vs. Prediction

Aspect	Training	Prediction
Client Filter	2+ policies	Exactly 1 policy
Target Variable	`second_product_category` (known)	`second_product_category` (predicted)
Purpose	Learn patterns	Apply learned patterns

9.3 Preprocessing Consistency

Critical: Prediction preprocessing must match training preprocessing exactly:

- Same feature engineering steps
- Same categorical encodings (using saved `categorical_mappings`)
- Same missing value imputation
- Same feature column order

Implementation: `final_preprocessor.py` contains shared preprocessing functions used by both training and prediction.

9.4 Model Deployment

Deployment Method: MLflow PythonModel (`PreprocessAndPredictModel`) **Components**:

1. **Data Loading**: From Unity Catalog table
2. **Preprocessing**: Using `preprocess_for_prediction()` function

3. **Prediction**: Using trained LightGBM model
4. **Post-processing**: Convert predictions to product names using `id2prod`

9.5 Enhanced Talking Points

Purpose: Generate human-readable explanations for predictions **Method**: SHAP (SHapley Additive exPlanations) analysis **Output**: Top 5 features contributing to each prediction with contextual talking points

Templates: Defined in `ENHANCED_TEMPLATES` dictionary, matching the feature set from training (cell 4).

---

10. Summary Statistics

10.1 Data Flow Summary

```
 Source Table (wealth_management_client_metrics)
    ↓
Remove Duplicates
    ↓
Create product_category
    ↓
Filter: Active policies, non-null fields
    ↓
Order by register_date per client
    ↓
Filter: Clients with 2+ policies
    ↓
Separate first and second policies
    ↓
Join first + second policy data
    ↓
Feature Engineering (allocation ratios, temporal features)
    ↓
Create labels (second_product_category → integer)
    ↓
Impute missing values
    ↓
Encode categorical features
    ↓
Split: 80% train / 10% val / 10% test
    ↓
Convert to Pandas
    ↓
Train LightGBM model
    ↓
Hyperparameter tuning (Optuna)
    ↓
Register best model to MLflow
```

10.2 Key Numbers

- **Training Records**: ~213,597
- **Features**: 25
- **Classes**: 7 product categories

---

11. Best Practices and Notes

11.1 Reproducibility

- Saved all artifacts (mappings, encodings) for consistent preprocessing
- Version control model artifacts and preprocessing code

Appendix: Code References

Key Files

1. **Training Notebook**: `1. Data extraction and model training.ipynb`
    - Cell 4: Main preprocessing and training logic
    - Cell 6: Hyperparameter tuning
    - Cell 7: Model registration
    - Cell 18: SHAP analysis and talking points
2. **Preprocessing Module**: `final_preprocessor.py`
    - `preprocess_for_training()` : Training data preprocessing
    - `preprocess_for_prediction()` : Prediction data preprocessing
    - Shared feature engineering functions
3. **Prediction Model**: `PreprocessAndPredictModel.py`
    - MLflow PythonModel wrapper
    - Handles data loading, preprocessing, and prediction
4. **Prediction Pipeline**: `2. final_pipeline.py`
    - End-to-end prediction script
    - SHAP analysis and talking points generation
    - Saving the predictions with demographics in the table 'eda_smartlist.us_wealth_management_smartlist.ML_predictions_single_policy'

Product Category Mapping Rules

B.1 LIFE_INSURANCE

- `prod_lob == "LIFE"`
- `sub_product_level_1` in: `["VLI", "WL", "UL/IUL", "TERM", "PROTECTIVE PRODUCT"]`
- `sub_product_level_2` contains "LIFE"
- `sub_product_level_2` in: `["VARIABLE UNIVERSAL LIFE", "WHOLE LIFE", "UNIVERSAL LIFE", "INDEX UNIVERSAL LIFE", "TERM PRODUCT", "VARIABLE LIFE", "SURVIVORSHIP WHOLE LIFE", "MONY PROTECTIVE PRODUCT"]`

B.2 RETIREMENT

- `prod_lob` in: `["GROUP RETIREMENT", "INDIVIDUAL RETIREMENT"]`
- `sub_product_level_1` in: `["EQUIVEST", "RETIREMENT 401K", "ACCUMULATOR", "RETIREMENT CORNERSTONE", "SCS", "INVESTMENT EDGE"]`
- `sub_product_level_2` contains: "403B", "401", "IRA", or "SEP"

B.3 INVESTMENT

- `prod_lob == "BROKER DEALER"`
- `sub_product_level_1` in: `["INVESTMENT PRODUCT - DIRECT", "INVESTMENT PRODUCT - BROKERAGE", "INVESTMENT PRODUCT - ADVISORY", "DIRECT", "BROKERAGE", "ADVISORY", "CASH SOLICITOR"]`
- `sub_product_level_2` contains: "Investment", "Brokerage", or "Advisory"

B.4 NETWORK_PRODUCTS

- `prod_lob == "NETWORK"`
- `sub_product_level_1 == "NETWORK PRODUCTS"`
- `sub_product_level_2 == "NETWORK PRODUCTS"`

B.5 DISABILITY

- `prod_lob == "OTHERS"` AND `sub_product_level_1 == "HAS"`
- `sub_product_level_2 == "HAS - DISABILITY"`

B.6 HEALTH

- `prod_lob == "OTHERS"` (and not DISABILITY)
- `sub_product_level_2 == "GROUP HEALTH PRODUCTS"`

B.7 OTHER

- All products not matching above rules