Complete Project Comparison: Main_project vs novartis_datathon_2025-Arman

Executive Summary

This document provides a comprehensive comparison between **Main_project** and **novartis_datathon_2025-Arman** for the Novartis Datathon 2025 generic erosion forecasting challenge.

Aspect	Main_project	novartis_datathon_2025-Arman
Philosophy	Rapid prototyping, simplicity	Production-ready, enterprise-grade
Configuration	Python constants	YAML-driven
Models	4 model families	10+ model families
Testing	None	Full pytest suite
Deep Learning	Placeholder only	CNN-LSTM, KG-GCN-LSTM
Experiment Tracking	None	MLflow/W&B integration

1. Project Architecture

1.1 Directory Structure

Main_project

```
Main_project/
├── data/
│   ├── raw/            # Input CSVs
│   └── processed/      # Intermediate data
├── src/
│   ├── config.py                # All settings
│   ├── data_loader.py           # Load raw data
│   ├── bucket_calculator.py     # Bucket/avg_vol calculation
│   ├── feature_engineering.py   # Feature creation
│   ├── models.py                # All model classes
│   ├── evaluation.py            # CV and evaluation
│   ├── metric_calculation.py    # Official metrics
│   ├── pipeline.py              # End-to-end pipeline
│   ├── submission.py            # Generate submissions
│   ├── eda_analysis.py          # Exploratory analysis
│   ├── external_data.py         # External data (stubs)
│   ├── visibility_sources.py    # Visibility features
│   ├── scenarios/               # Scenario definitions
│   └── training/                # Training utilities
├── scripts/
│   ├── run_pipeline.py
│   ├── train_models.py
│   └── generate_final_submissions.py
├── notebooks/         # Interactive analysis
├── models/            # Saved model files
├── submissions/       # Output submissions
└── reports/           # Figures and analysis
```

novartis_datathon_2025-Arman

```
novartis_datathon_2025-Arman/
├── configs/
│   ├── data.yaml                    # Data paths & schema
│   ├── features.yaml                # Feature engineering settings
│   ├── run_defaults.yaml            # Reproducibility & validation
│   ├── model_cat.yaml               # CatBoost hyperparameters
│   ├── model_lgbm.yaml              # LightGBM hyperparameters
│   ├── model_xgb.yaml               # XGBoost hyperparameters
│   ├── model_linear.yaml            # Linear model settings
│   ├── model_nn.yaml                # Neural network settings
│   ├── model_cnn_lstm.yaml          # CNN-LSTM settings
│   ├── model_kg_gcn_lstm.yaml       # Graph neural network
│   ├── model_lstm.yaml              # Pure LSTM settings
│   └── model_hybrid.yaml            # Physics-ML hybrid
├── src/
│   ├── utils.py                     # Seeding, logging, config
│   ├── data.py                      # Data loading & panel
│   ├── features.py                  # Feature engineering
│   ├── evaluate.py                  # Official metric wrappers
│   ├── validation.py                # Series-level splits
│   ├── train.py                     # CLI training pipeline
│   ├── inference.py                 # Prediction & submission
│   ├── config_sweep.py              # Grid search expansion
│   ├── sequence_builder.py          # Deep learning sequences
│   ├── graph_utils.py               # Drug graph construction
│   ├── external_data.py             # External data sources
│   ├── visibility_sources.py        # Supply chain features
│   └── models/
│       ├── base.py                  # Abstract interface
│       ├── cat_model.py             # CatBoost
│       ├── lgbm_model.py            # LightGBM
│       ├── xgb_model.py             # XGBoost
│       ├── linear.py                # Ridge/Lasso/ElasticNet
│       ├── nn.py                    # MLP neural network
│       ├── cnn_lstm.py              # CNN-LSTM
│       ├── kg_gcn_lstm.py           # Knowledge Graph GCN-LSTM
│       ├── gcn_layers.py            # GCN layer implementations
│       ├── arihow.py                # ARIMA + Holt-Winters
│       ├── hybrid_physics_ml.py     # Physics + ML hybrid
│       ├── ensemble.py              # Ensemble methods
│       └── baselines.py             # Simple baselines
│
```

```
    ├── tests/
    │   ├── test_smoke.py            # Comprehensive tests
    │   └── pytest.ini               # Test configuration
    ├── artifacts/         # Model outputs & logs
    ├── submissions/       # Output submissions
    ├── docs/              # Documentation
    └── env/               # Environment files
```

Pros - Main_project:

- ☑ Flat structure is easy to navigate
- ☑ Single config file reduces complexity
- ☑ Quick to get started

Cons - Main_project:

- ✖ All models in one 1,800+ line file
- ✖ No separation of concerns
- ✖ Harder to maintain at scale

Pros - Arman:

- ☑ Clear separation of concerns
- ☑ Each model in its own file
- ☑ Configuration separate from code
- ☑ Test suite for validation

Cons - Arman:

- ✖ More files to navigate
- ✖ Steeper learning curve
- ✖ Configuration overhead for small experiments

---

2. Configuration System

2.1 Main_project Approach

All configuration in `src/config.py`:

```
# Hard-coded constants
DEFAULT_DECAY_RATE = 0.03
N_GXS_CAP_PERCENTILE = 99
LIGHTGBM_PARAMS = {
    'objective': 'regression',
    'metric': 'mae',
    'n_estimators': 500,
    'learning_rate': 0.05,
    'max_depth': 8,
    # ...
}

# Multi-config mode (grid search)
MULTI_CONFIG_PARAMETERS = [
    {'id': 'default', 'DEFAULT_DECAY_RATE': 0.03},
    {'id': 'high_decay', 'DEFAULT_DECAY_RATE': 0.05},
    {'id': 'fast_decay', 'DEFAULT_DECAY_RATE': 0.05},
]
```

Pros:

- ☑ All settings in one place
- ☑ Type checking by Python
- ☑ IDE autocomplete works
- ☑ No parsing overhead

Cons:

- ✗ Code changes required for config changes
- ✗ No version control of configs separate from code
- ✗ Grid search requires code modification
- ✗ Hard to reproduce past experiments

2.2 novartis_datathon_2025-Arman Approach

YAML configuration files:

```yaml
# configs/run_defaults.yaml
reproducibility:
  seed: 42
  deterministic: true

validation:
  val_fraction: 0.2
  stratify_by: "bucket"
  split_level: "series"

sample_weights:
  scenario1:
    months_0_5: 3.0
    months_6_11: 1.5
    months_12_23: 1.0
```

```yaml
# configs/model_cat.yaml
model:
  type: catboost
  hyperparameters:
    iterations: [500, 1000]      # Lists auto-expand to sweeps
    learning_rate: [0.03, 0.05]
    depth: [6, 8]
    l2_leaf_reg: 3
```

Pros:

- ✅ Configuration separate from code
- ✅ Version control of configs
- ✅ Lists auto-expand to sweeps
- ✅ Reproducibility via config snapshots
- ✅ No code changes for experiments

Cons:

- ❌ YAML parsing overhead
- ❌ No type checking
- ❌ IDE support limited
- ❌ Config files can become complex

---

3. Data Pipeline

3.1 Data Loading

Main_project

```python
# src/data_loader.py
def load_all_data():
    """Load all raw CSV files."""
    volume_train = pd.read_csv(DATA_RAW / "df_volume_train.csv")
    generics_train = pd.read_csv(DATA_RAW / "df_generics_train.csv")
    medicine_info = pd.read_csv(DATA_RAW / "df_medicine_info_train.csv")
    return volume_train, generics_train, medicine_info
```

Pros:

- ✅ Simple and direct
- ✅ Easy to understand
- ✅ Fast for small datasets

Cons:

- ❌ No schema validation
- ❌ No caching

- ✗ No duplicate checking

novartis_datathon_2025-Arman

```
# src/data.py
def load_raw_data(data_config: dict, split: str = 'train') -> Dict[str,
pd.DataFrame]:
    """
    Load raw data with validation and caching.

    - Validates column schema against config
    - Checks for duplicates
    - Normalizes data types
    - Caches to Parquet for faster reloads
    """
    # Schema validation
    validate_schema(df, expected_columns)

    # Duplicate checking
    check_duplicates(df, key_cols)

    # Cache to Parquet
    if use_cache:
        df.to_parquet(cache_path)
```

Pros:

- ☑ Schema validation catches data issues early
- ☑ Parquet caching for faster reloads
- ☑ Duplicate detection
- ☑ Type normalization

Cons:

- ✗ More complex implementation
- ✗ Initial cache build takes time
- ✗ Config dependency

3.2 Panel Construction

Feature	Main_project	Arman
Key columns	`country`, `brand_name`, `months_postgx`	Same
Schema validation	✗	☑
Duplicate handling	Manual	Automated
Missing value imputation	Basic fillna	Strategy-based

Feature	Main_project	Arman
Caching	✗	Parquet-based

4. Model Implementations

4.1 Available Models

Model Family	Main_project	Arman
Baselines	☑ Naive, Linear/Exp Decay	☑ + Historical Curve, Trend
LightGBM	☑	☑ (separate file)
XGBoost	☑	☑ (separate file)
CatBoost	☑	☑ (separate file)
Linear	☑ Ridge, Lasso, ElasticNet	☑ + Huber, Polynomial
MLP Neural Network	☑ MLPRegressor	☑ PyTorch MLP
ARIMA/Holt-Winters	☑ ARIHOWModel	☑ ARIHOWModel
Hybrid Physics-ML	☑ HybridPhysicsMLModel	☑ HybridPhysicsMLModel
CNN-LSTM	✗ Placeholder	☑ Full implementation
KG-GCN-LSTM	✗	☑ Knowledge Graph GCN
Pure LSTM	✗ Placeholder	☑ Full implementation

4.2 Model Interface

Main_project (Ad-hoc)

```
class GradientBoostingModel:
    def __init__(self, model_type='lightgbm', **params):
        self.model_type = model_type
        self.params = params

    def fit(self, X_train, y_train, X_val=None, y_val=None):
        # Different logic per model type

    def predict(self, X):
        return self.model.predict(X)

    def save(self, name):
        joblib.dump(self.model, path)
```

Pros:

- ☑ Simple to implement
- ☑ Flexible

Cons:

- ✗ No standard interface
- ✗ Feature importance varies by model
- ✗ No factory pattern

novartis_datathon_2025-Arman (Protocol/Interface)

```python
# src/models/base.py
class BaseModel(ABC):
    """Abstract base class for all models."""

    @abstractmethod
    def fit(self, X_train, y_train, X_val=None, y_val=None,
            sample_weight=None, **kwargs) -> 'BaseModel':
        pass

    @abstractmethod
    def predict(self, X: pd.DataFrame) -> np.ndarray:
        pass

    @abstractmethod
    def save(self, path: Path) -> None:
        pass

    @classmethod
    @abstractmethod
    def load(cls, path: Path) -> 'BaseModel':
        pass

    def get_feature_importance(self) -> Optional[pd.DataFrame]:
        pass

# Factory pattern
MODEL_REGISTRY = {
    'catboost': CatBoostModel,
    'lightgbm': LGBMModel,
    'xgboost': XGBModel,
    'linear': LinearModel,
    'nn': NNModel,
    'cnn_lstm': CNNLSTMModel,
    'kg_gcn_lstm': KGGCNLSTMModel,
}

def get_model(model_type: str, **kwargs) -> BaseModel:
    return MODEL_REGISTRY[model_type](**kwargs)
```

Pros:

- ☑ Consistent interface across all models
- ☑ Factory pattern for easy instantiation
- ☑ Easy to add new models
- ☑ Lazy loading for optional dependencies

Cons:

- ✗ More boilerplate per model
- ✗ Abstract methods must be implemented

4.3 Deep Learning Models

Main_project

```
class LSTMModelPlaceholder:
    """
    Placeholder for LSTM model.
    Dependency-heavy (PyTorch/TF); intentionally minimal.
    """
    def __init__(self):
        raise NotImplementedError("LSTM not implemented")
```

novartis_datathon_2025-Arman

```
# src/models/cnn_lstm.py
class CNNLSTMModel(BaseModel):
    """
    CNN-LSTM for drug sales prediction (Li et al. 2024).

    Architecture:
    - 1D Conv layers for local pattern extraction
    - LSTM layers for temporal dependencies
    - Dense layers for final prediction
    """
    def __init__(self, input_dim, seq_len, hidden_dim=64,
                 n_lstm_layers=2, dropout=0.2):
        self.model = nn.Sequential(
            nn.Conv1d(input_dim, 32, kernel_size=3),
            nn.ReLU(),
            nn.LSTM(32, hidden_dim, n_lstm_layers, dropout=dropout),
            nn.Linear(hidden_dim, 1)
        )

# src/models/kg_gcn_lstm.py
class KGGCNLSTMModel(BaseModel):
    """
    Knowledge Graph GCN + LSTM (KG-GCN-LSTM paper).

    Architecture:
    - Drug knowledge graph with therapeutic area relationships
    - Graph Convolutional Network for drug embeddings
    - LSTM for temporal forecasting
    """
```

Summary:

- Main_project: Deep learning is a placeholder - not functional
- Arman: Full implementations of CNN-LSTM and KG-GCN-LSTM

---

5. Training Pipeline

5.1 Main_project Training

```python
# scripts/train_models.py
def train_model(scenario, model_type):
    # Load data
    data = load_all_data()

    # Create features
    features = create_all_features(data, scenario)

    # Split
    X_train, X_val, y_train, y_val = split_data(features)

    # Train
    model = GradientBoostingModel(model_type)
    model.fit(X_train, y_train, X_val, y_val)

    # Evaluate
    metrics = evaluate_model(model, X_val, y_val)

    return model, metrics
```

Pros:

- ☑ Simple and linear
- ☑ Easy to follow
- ☑ Quick to modify

Cons:

- ✗ No CLI interface

- ❌ No experiment tracking
- ❌ No hyperparameter optimization
- ❌ No checkpointing

5.2 novartis_datathon_2025-Arman Training

```python
# src/train.py - CLI entry point
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--scenario', type=int, choices=[1, 2])
    parser.add_argument('--model', type=str, default='catboost')
    parser.add_argument('--cv', action='store_true')
    parser.add_argument('--n-folds', type=int, default=5)
    parser.add_argument('--hpo', action='store_true')
    parser.add_argument('--hpo-trials', type=int, default=50)
    parser.add_argument('--sweep', action='store_true')
    parser.add_argument('--parallel', action='store_true')
    parser.add_argument('--enable-tracking', action='store_true')
    # ...

# Example commands:
# python -m src.train --scenario 1 --model catboost
# python -m src.train --scenario 1 --model catboost --cv --n-folds 5
# python -m src.train --scenario 1 --model catboost --hpo --hpo-trials 50
# python -m src.train --full-pipeline --model catboost --parallel
```

Features:

- ☑ Full CLI with argparse
- ☑ Cross-validation (K-fold, stratified)
- ☑ Hyperparameter optimization (Optuna)
- ☑ Config sweep expansion
- ☑ Parallel training
- ☑ Experiment tracking (MLflow/W&B)
- ☑ Checkpointing
- ☑ Memory profiling

Pros:

- ☑ Reproducible experiments
- ☑ Automated hyperparameter search
- ☑ Experiment comparison
- ☑ Production-ready

Cons:

- ❌ Complex to understand
- ❌ Many CLI options

- ❌ Slower for quick experiments

---

6. Validation Strategy

6.1 Main_project

```python
# src/evaluation.py
def split_train_validation(df, val_fraction=0.2):
    """Simple random split."""
    train_idx = np.random.choice(len(df), int(len(df) * (1 - val_fraction)))
    val_idx = np.setdiff1d(np.arange(len(df)), train_idx)
    return df.iloc[train_idx], df.iloc[val_idx]
```

Issues:

- ⚠ Row-level split can leak information
- ⚠ Same brand may appear in train and validation
- ⚠ Not stratified by bucket

6.2 novartis_datathon_2025-Arman

```python
# src/validation.py
def create_validation_split(panel, val_fraction=0.2, stratify_by='bucket'):
    """
    Series-level split with stratification.

    CRITICAL: Split by (country, brand_name), not by row.
    This prevents data leakage where future months inform past predictions.
    """
    series_ids = panel.groupby(['country', 'brand_name']).ngroup()
    unique_series = panel.drop_duplicates(['country', 'brand_name'])

    # Stratified split by bucket
    train_series, val_series = train_test_split(
        unique_series,
        test_size=val_fraction,
        stratify=unique_series['bucket'],
        random_state=42
    )

    train_mask = series_ids.isin(train_series.index)
    return panel[train_mask], panel[~train_mask]

def simulate_scenario(panel, scenario):
    """
    Simulate test-time constraints.
    Scenario 1: Only pre-entry data available
    Scenario 2: First 6 months available
    """
    if scenario == 1:
        return panel[panel['months_postgx'] < 0]
    else:
        return panel[panel['months_postgx'] < 6]
```

Features:

- ☑ Series-level split (no leakage)
- ☑ Stratification by bucket
- ☑ Scenario simulation
- ☑ Adversarial validation
- ☑ Out-of-fold prediction handling

---

7. Experiment Tracking

7.1 Main_project

No experiment tracking. Results printed to console.

7.2 novartis_datathon_2025-Arman

```
# src/train.py
class ExperimentTracker:
    """Unified experiment tracking with MLflow or W&B."""

    def __init__(self, backend='mlflow', experiment_name='novartis'):
        self.backend = backend
        if backend == 'mlflow':
            mlflow.set_experiment(experiment_name)
        elif backend == 'wandb':
            wandb.init(project=experiment_name)

    def log_params(self, params: dict):
        if self.backend == 'mlflow':
            mlflow.log_params(params)
        elif self.backend == 'wandb':
            wandb.config.update(params)

    def log_metrics(self, metrics: dict, step=None):
        # ...

    def log_artifact(self, path):
        # ...
```

Features:

- [x] MLflow integration
- [x] Weights & Biases integration
- [x] Config snapshots
- [x] Git hash capture
- [x] Artifact logging

---

8. Ensemble Methods

8.1 Main_project

Simple weighted blending:

```python
def blend_predictions(preds_list, weights):
    """Weighted average of predictions."""
    return np.average(preds_list, axis=0, weights=weights)
```

8.2 novartis_datathon_2025-Arman

```python
# src/models/ensemble.py
class AveragingEnsemble(BaseModel):
    """Simple averaging of predictions."""

class WeightedAveragingEnsemble(BaseModel):
    """Weighted average with optimizable weights."""

class StackingEnsemble(BaseModel):
    """Two-level stacking with meta-learner."""

class BlendingEnsemble(BaseModel):
    """Blending with holdout predictions."""

def optimize_ensemble_weights(models, X_val, y_val, metric_fn):
    """Find optimal weights using scipy.optimize."""
    from scipy.optimize import minimize

    def objective(weights):
        preds = sum(w * m.predict(X_val) for w, m in zip(weights, models))
        return metric_fn(y_val, preds)

    result = minimize(objective, x0=np.ones(len(models))/len(models))
    return result.x
```

---

9. Testing

9.1 Main_project

No automated tests.

9.2 novartis_datathon_2025-Arman

```python
# tests/test_smoke.py
class TestDataLoading:
    def test_load_raw_data(self):
        """Test that raw data loads correctly."""

    def test_schema_validation(self):
        """Test that schema validation works."""

class TestFeatures:
    def test_no_leakage(self):
        """Test that features don't leak future information."""

    def test_scenario_cutoffs(self):
        """Test that scenario cutoffs are enforced."""

class TestModels:
    def test_catboost_fit_predict(self):
        """Test CatBoost can fit and predict."""

    def test_model_save_load(self):
        """Test model serialization."""

class TestMetrics:
    def test_metric1_official(self):
        """Test Metric 1 matches official implementation."""

    def test_metric2_official(self):
        """Test Metric 2 matches official implementation."""

class TestIntegrationEndToEnd:
    def test_end_to_end_pipeline(self):
        """Test full pipeline from data to submission."""
```

Coverage:

- ☑ Import tests
- ☑ Config alignment tests
- ☑ Data loading tests
- ☑ Feature leakage tests
- ☑ Model training tests
- ☑ Metric calculation tests

- ☑ End-to-end integration tests

---

10. Reproducibility

10.1 Main_project

```
# Manual seed setting
np.random.seed(42)
```

10.2 novartis_datathon_2025-Arman

```
# src/utils.py
def set_seed(seed: int = 42):
    """Set all random seeds for reproducibility."""
    import random
    import numpy as np
    import torch

    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True

    # Set environment variables
    os.environ['PYTHONHASHSEED'] = str(seed)

def compute_config_hash(configs: dict) -> str:
    """Generate hash for configuration reproducibility."""
    config_str = json.dumps(configs, sort_keys=True)
    return hashlib.md5(config_str.encode()).hexdigest()[:8]

def save_config_snapshot(artifacts_dir: Path, configs: dict):
    """Save configuration snapshot for reproducibility."""
    snapshot = {
        'timestamp': datetime.now().isoformat(),
        'git_hash': get_git_hash(),
        'configs': configs
    }
    with open(artifacts_dir / 'config_snapshot.json', 'w') as f:
        json.dump(snapshot, f, indent=2)
```

Features:

- ☑ Comprehensive seed setting (Python, NumPy, PyTorch)

- [x] Config hashing
- [x] Git hash capture
- [x] `reproduce.sh` script

---

11. Inference & Submission

11.1 Main_project

```python
# src/submission.py
def generate_submission(model, test_data, avg_j_df):
    """Generate submission file."""
    predictions = model.predict(test_features)

    # Denormalize
    predictions = predictions * avg_j_df['avg_vol']

    # Format
    submission = pd.DataFrame({
        'country': test_data['country'],
        'brand_name': test_data['brand_name'],
        'months_postgx': test_data['months_postgx'],
        'volume': predictions
    })

    return submission
```

11.2 novartis_datathon_2025-Arman

```python
# src/inference.py
def generate_submission(
    model,
    test_panel,
    pre_entry_stats,
    scenario: int,
    fallback_strategy: str = 'exponential_decay'
) -> pd.DataFrame:
    """
    Generate submission with edge case handling.

    Features:
    - Automatic scenario detection
    - Edge case fallback (missing series, outliers)
    - Volume clipping to valid range
    - Format validation
    """
    predictions = model.predict(test_features)
```

```
        # Denormalize
        predictions = predictions * pre_entry_stats['avg_vol_12m']

        # Apply fallback for edge cases
        predictions = apply_edge_case_fallback(
            predictions,
            test_panel,
            strategy=fallback_strategy
        )

        # Validate format
        validate_submission_format(submission, template)

        return submission

    def detect_test_scenarios(test_panel) -> Dict[str, List[str]]:
        """Detect which series belong to Scenario 1 vs Scenario 2."""
        # Series with months 0-5 data → Scenario 2
        # Series without → Scenario 1
```

---

12. Summary: When to Use Each Project

Use Main_project When:

1. **Rapid Prototyping**: You need to quickly test an idea
2. **Learning**: You're new to the competition and want to understand the pipeline
3. **Simple Models**: You're focusing on tree-based models only
4. **Small Team**: One person maintains the codebase
5. **Limited Time**: Competition deadline is near
6. **Debugging**: You need to trace through code easily

Use novartis_datathon_2025-Arman When:

1. **Production Deployment**: You need robust, tested code
2. **Deep Learning**: You want to use CNN-LSTM or graph neural networks
3. **Experimentation at Scale**: You need automated hyperparameter search
4. **Team Collaboration**: Multiple people work on the codebase
5. **Reproducibility**: You need to reproduce past experiments exactly
6. **Competition Winning**: You're aiming for top rankings and need every edge

---

13. Recommendations for Best Results

Hybrid Approach

Consider combining the best of both:

1. **From Main_project:**

- Simple, flat structure for quick experiments
- Integrated sample weighting
- Therapeutic area erosion rankings

2. **From Arman:**

- YAML configuration for reproducibility
- Proper series-level validation
- Deep learning models for ensembling
- Experiment tracking
- Test suite for validation

Quick Wins

1. Add series-level validation to Main_project
2. Use Arman's seasonal features
3. Implement Arman's future n_gxs features (if allowed)
4. Add basic pytest tests to Main_project
5. Use Arman's ensemble optimization

Appendix A: Code Metrics

Metric	Main_project	Arman
Total Python lines	~5,000	~15,000
Number of .py files	~15	~30
Config files	1 (Python)	12 (YAML)
Test files	0	1 (5,700+ lines)
Documentation files	~15	~10
Model implementations	4	12

Appendix B: Dependencies

Main_project

```
numpy
pandas
scikit-learn
lightgbm
xgboost
catboost
statsmodels
joblib
matplotlib
seaborn
```

novartis_datathon_2025-Arman

```
# Core
numpy
pandas
scikit-learn
lightgbm
xgboost
catboost
statsmodels
joblib

# Deep Learning (optional)
torch
torch-geometric

# Experiment Tracking (optional)
mlflow
wandb

# HPO (optional)
optuna

# Testing
pytest

# Utilities
pyyaml
tqdm
```

Document created: November 29, 2025 Last updated: November 29, 2025