**Family Linkage Pipeline Documentation**

This document provides detailed technical documentation for both the training and testing pipelines in the Family Linkage Models project.

**Training Pipeline Documentation**

**Overview**

The training pipeline (training\_pipeline/main.py) processes labeled datasets to create machine learning models for relationship detection. It combines positive relationship examples with negative examples generated through record comparison to train Random Forest classifiers.

**Architecture Flow**

Raw Data → Normalization → Database Loading → Record Comparison →

Feature Generation → Positive Pair Processing → Model Training →

Evaluation → Model Persistence

**Detailed Code Analysis**

**1. Initialization and Setup**

def setup\_logging():

"""Setup logging configuration"""

os.makedirs('logs', exist\_ok=True)

logging.basicConfig(

filename='logs/training\_app.log',

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'

)

return logging.getLogger(\_\_name\_\_)

**Purpose**: Configures centralized logging for the training process. All operations, errors, and performance metrics are logged to logs/training\_app.log.

**Key Features**:

* Creates logs directory if it doesn't exist
* Uses INFO level logging to capture detailed process information
* Structured format includes timestamp, logger name, level, and message

**2. Configuration Management**

def load\_config(config\_path='config.yaml'):

"""Load configuration from YAML file"""

with open(config\_path, 'r') as file:

return yaml.safe\_load(file)

**Purpose**: Loads all configuration parameters from YAML file including database connections, file paths, and model hyperparameters.

**3. Data Loading and Preprocessing**

def read\_and\_clean\_data(config, relationship\_type, logger):

"""Read and clean training data"""

try:

# Read dataset

dataset = pd.read\_csv(

config['data']['raw\_dataset'],

usecols=['id', 'last\_name', 'phone', 'middle\_name', 'zip',

'city', 'dob', 'state', 'address', 'sex', 'ssn'],

dtype='string'

)

# Read and filter labels

labels = pd.read\_csv(config['data']['raw\_labels'], dtype='string')

filtered\_labels = labels[labels['relationship'] == relationship\_type]

# Normalize data using common module

cleaned\_dataset = normalize(dataset, logger)

return cleaned\_dataset, filtered\_labels

**Purpose**: Loads raw training data and relationship labels, then applies normalization.

**Key Operations**:

* **Column Selection**: Only loads necessary columns to optimize memory usage
* **Label Filtering**: Extracts only the relationships relevant to the current training session
* **Data Type Consistency**: Forces string dtype to handle mixed data types
* **Normalization**: Applies advanced text cleaning, removes diacritics, handles missing values

**Normalization Process** (from common.py):

* Unicode normalization and diacritical mark removal
* Advanced name cleaning with suffix/prefix removal
* Placeholder record removal (BABY, INFANT, VOID, UNKNOWN)
* Date standardization and SSN cleaning

**4. Database Initialization**

def setup\_database(config, cleaned\_dataset, labels, logger, project\_root):

"""Setup database with cleaned data"""

try:

# Create engine

db\_config = config['database']

db\_string = f"postgresql://{db\_config['user']}:{db\_config['password']}@{db\_config['host']}:{db\_config['port']}/{db\_config['database']}"

engine = create\_engine(db\_string)

# Initialize tables and functions

tables\_sql\_path = os.path.join(project\_root, 'scripts', 'postgres\_tables.sql')

functions\_sql\_path = os.path.join(project\_root, 'scripts', 'postgres\_functions.sql')

# Execute SQL scripts

with open(tables\_sql\_path, 'r') as file:

sql\_commands = file.read()

with engine.connect() as connection:

connection.execute(text(sql\_commands))

with open(functions\_sql\_path, 'r') as file:

sql\_commands = file.read()

with engine.connect().execution\_options(autocommit=True) as connection:

connection.execute(text(sql\_commands))

**Purpose**: Initializes PostgreSQL database with required tables and functions, then loads cleaned data.

**Database Setup Process**:

1. **Connection Pool Creation**: Establishes database connection with connection pooling
2. **Schema Initialization**: Creates records and labels tables
3. **Function Installation**: Installs all PostgreSQL functions for comparison operations
4. **Data Loading**: Bulk loads normalized data with proper column types
5. **Index Creation**: Automatically creates indexes for performance

**5. Record Comparison Engine**

compare(

database\_url=db\_string,

job\_schema='public',

records\_table='records',

logger=logger,

size\_threshold=size\_threshold,

max\_block\_size=max\_block\_size,

window\_size=window\_size,

overlap=overlap,

blocking\_batch\_size=blocking\_batch\_size,

num\_workers=num\_workers

)

**Purpose**: Generates negative training examples by comparing all record pairs using advanced blocking techniques.

**Comparison Strategies**:

**Small Datasets (≤ size\_threshold)**:

* **Exhaustive Comparison**: Compares every record against every other record

**Large Datasets (> size\_threshold)**:

* **Optimized Blocking**: Uses multiple blocking strategies to reduce comparison space
* **Sliding Window**: Processes blocks in windows with overlap to ensure completeness

**Blocking Strategies**:

1. **Name-based Blocking**: N\_[lastname\_soundex]\_[lastname\_prefix]
   * Groups records with similar name sounds and prefixes
   * Handles name variations and misspellings
2. **Address-based Blocking**: A\_[zip\_prefix]\_[address\_prefix]
   * Groups records from same geographic areas
   * Useful for finding family members in same household
3. **Demographic Blocking**: D\_[birth\_year]\_[sex]\_[lastname\_soundex]
   * Groups records with similar demographics
   * Effective for finding siblings or age-related relationships

**6. Feature Engineering**

The comparison process generates standardized features for machine learning:

-- Core similarity features

edit\_distance(r1.last\_name, r2.last\_name) as edit\_dist\_ln,

edit\_distance(r1.phone, r2.phone) as edit\_dist\_phone\_num,

edit\_distance(r1.middle\_name, r2.middle\_name) as edit\_dist\_mn,

edit\_distance(r1.zip, r2.zip) as edit\_dist\_zip,

edit\_distance(r1.city, r2.city) as edit\_dist\_city,

age\_difference(r1.dob, r2.dob) as age\_diff,

edit\_distance(r1.address, r2.address) as edit\_dist\_mail\_address,

-- Categorical match features

CASE WHEN r1.sex != r2.sex THEN 1 ELSE 0 END as sex\_diff,

CASE WHEN r1.ssn = r2.ssn THEN 1 ELSE 0 END as ssn\_match,

CASE WHEN r1.state = r2.state THEN 1 ELSE 0 END as state\_match,

-- Encoded demographic features

CASE WHEN r1.sex = 'M' THEN 0 WHEN r1.sex = 'F' THEN 1 ELSE -1 END as record1\_sex,

CASE WHEN r2.sex = 'M' THEN 0 WHEN r2.sex = 'F' THEN 1 ELSE -1 END as record2\_sex,

CASE WHEN age < 18 THEN 0 WHEN age BETWEEN 18 AND 50 THEN 1 ELSE 2 END as record1\_agecategory,

CASE WHEN age < 18 THEN 0 WHEN age BETWEEN 18 AND 50 THEN 1 ELSE 2 END as record2\_agecategory

**Feature Categories**:

* **Text Similarity**: Edit distance for names, addresses, geographic data
* **Temporal Features**: Age differences with normalization
* **Binary Indicators**: Exact matches for SSN, state, sex differences
* **Categorical Encodings**: Demographic categories for both records

**7. Positive Example Processing**

# Process positive pairs

with engine.connect().execution\_options(autocommit=True) as connection:

connection.execute(text("SELECT process\_positive\_record\_pairs()"))

**Purpose**: Generates feature vectors for known positive relationship pairs from the labels dataset.

**Process**:

1. Joins labels table with records table twice (for both from\_id and to\_id)
2. Computes identical feature set as comparison process
3. Stores results in processed\_positive\_records table
4. Ensures feature consistency between positive and negative examples

**8. Dataset Assembly and Preparation**

# Extract and combine data

processed\_records\_df = pd.read\_sql\_query("SELECT \* FROM processed\_records", engine)

processed\_records\_df['relationship'] = 0 # Negative examples

processed\_positive\_df = pd.read\_sql\_query("SELECT \* FROM processed\_positive\_records", engine)

processed\_positive\_df['relationship'] = 1 # Positive examples

combined\_df = pd.concat([processed\_records\_df, processed\_positive\_df], ignore\_index=True)

**Purpose**: Combines negative examples (from comparison) with positive examples (from labels) into a balanced training dataset.

**Data Processing Steps**:

1. **Label Assignment**: Negative examples = 0, Positive examples = 1
2. **ID Normalization**: Handles different ID formats (string vs integer)
3. **Deduplication**: Removes duplicate pairs, prioritizing positive labels
4. **Type Conversion**: Ensures consistent data types across all features

**9. Machine Learning Pipeline**

**Purpose**: Trains Random Forest classifier with hyperparameter optimization and stratified sampling.

**Model Training Features**:

* **Stratified Split**: Maintains class balance in train/test splits
* **Hyperparameter Configuration**: Loads model parameters from config file
* **Feature Selection**: Automatically excludes ID columns and similarity scores
* **Parallel Processing**: Uses all available CPU cores for training

**10. Model Evaluation and Visualization**

**Evaluation Metrics**:

* **Accuracy**: Overall classification accuracy
* **Classification Report**: Precision, recall, F1-score for both classes
* **ROC-AUC Score**: Area under ROC curve for probabilistic evaluation
* **Confusion Matrix**: True positives, false positives, etc.

**Visualization Outputs**:

1. **Feature Importance Plot**: Shows which features contribute most to predictions
2. **Confusion Matrix Heatmap**: Visual representation of classification performance
3. **ROC Curve**: Trade-off between true positive rate and false positive rate

**11. Model Persistence**

# Save trained model

model\_path = f"data/models/rf\_{relationship\_type}\_model.pkl"

joblib.dump(rf\_model, model\_path)

**Purpose**: Serializes trained model for use in testing pipeline.

**Testing Pipeline Documentation**

**Overview**

The testing pipeline (test\_pipeline/main.py) applies trained models to unlabeled datasets for relationship prediction. It focuses on memory efficiency and scalability for large datasets.

**Architecture Flow**

Test Data → Normalization → Database Loading → Record Comparison →

Feature Generation → Chunked Prediction → Results Output

**Detailed Code Analysis**

**1. Enhanced Progress Tracking**

class ProgressTracker:

"""Enhanced progress tracker for the testing pipeline"""

def \_\_init\_\_(self, total\_steps=8):

self.total\_steps = total\_steps

self.current\_step = 0

self.start\_time = time.time()

def update(self, step\_name):

self.current\_step += 1

progress = (self.current\_step / self.total\_steps) \* 100

elapsed = time.time() - self.start\_time

print(f"\n[Progress {progress:3.0f}%] {step\_name} ✓")

print(f"[Elapsed: {elapsed:.1f}s]")

return progress

**Purpose**: Provides real-time progress monitoring with elapsed time tracking for long-running operations.

**Features**:

* Step-by-step progress percentage
* Elapsed time tracking
* Visual progress indicators
* Console-based status updates

**2. Model Validation**

def validate\_model\_exists(relationship\_type, model\_dir):

"""Validate that the required model exists"""

model\_path = os.path.join(model\_dir, f'rf\_{relationship\_type}\_model.pkl')

if not os.path.exists(model\_path):

raise FileNotFoundError(f"Model file not found: {model\_path}")

return model\_path

**Purpose**: Ensures required trained model exists before starting the testing process.

**Validation Process**:

* Constructs expected model file path
* Verifies file existence
* Provides clear error messages if model is missing
* Returns validated path for loading

**3. Memory-Efficient Data Processing**

def cleanup\_previous\_predictions(output\_dir, logger):

"""Clean up previous prediction results"""

try:

cleanup\_old\_files(output\_dir, 'predictions\_\*.csv', logger)

logger.info("Previous predictions cleaned up")

except Exception as e:

logger.warning(f"Error during prediction cleanup: {e}")

**Purpose**: Cleans up previous prediction results to prevent disk space issues and confusion.

**4. Chunked Prediction Engine**

def predict\_chunked(engine, job\_schema, relationship, model\_directory,

output\_directory, logger, chunk\_size=50000):

"""Generate predictions in memory-efficient chunks"""

# Load trained model

model\_path = os.path.join(model\_directory, f'rf\_{relationship}\_model.pkl')

model = joblib.load(model\_path)

# Process in chunks

for chunk\_df in extract\_processed\_records\_chunked(engine, job\_schema, chunk\_size, logger):

if len(chunk\_df) == 0:

continue

# Prepare features

feature\_columns = [col for col in chunk\_df.columns

if col not in ['from\_id', 'to\_id', 'relationship', 'similarity\_score']]

X = chunk\_df[feature\_columns]

# Make predictions

predictions = model.predict(X)

probabilities = model.predict\_proba(X)[:, 1]

# Create results for this chunk

chunk\_results = chunk\_df[['from\_id', 'to\_id']].copy()

chunk\_results[f'relationship\_{relationship}'] = predictions

chunk\_results[f'predicted\_probability\_{relationship}'] = probabilities

# Save results (append mode after first chunk)

chunk\_results.to\_csv(results\_path, mode='w' if first\_chunk else 'a',

header=first\_chunk, index=False)

**Purpose**: Processes large datasets in manageable chunks to prevent memory overflow.

**Chunked Processing Benefits**:

* **Memory Efficiency**: Never loads entire dataset into memory simultaneously
* **Scalability**: Can handle datasets larger than available RAM
* **Progress Tracking**: Shows progress as each chunk completes

**5. Probabilistic Output Generation**

# Make predictions

predictions = model.predict(X) # Binary classification

probabilities = model.predict\_proba(X)[:, 1] # Probability of positive class

# Create results

chunk\_results[f'relationship\_{relationship}'] = predictions

chunk\_results[f'predicted\_probability\_{relationship}'] = probabilities

**Purpose**: Generates both binary predictions and probability scores for decision making.

**6. Performance Optimization**

The testing pipeline includes several performance optimizations:

**Database Optimizations**:

* Connection pooling for efficient database access
* Batch processing for large result sets
* Index utilization for fast lookups

**Memory Management**:

* Chunked processing to control memory usage
* Garbage collection between chunks
* Streaming I/O for large files

**Parallel Processing**:

* Worker-based comparison for large datasets
* Parallel feature computation

**Common Components**

**1. Data Normalization (common.py)**

def normalize(dataframe, logger):

"""Advanced data normalization with comprehensive cleaning"""

# Text normalization

for col in text\_columns:

if col == 'last\_name':

df[col] = df[col].apply(\_clean\_name\_advanced)

else:

df[col] = df[col].apply(\_normalize\_string\_unified)

# Date standardization

if 'dob' in df.columns:

df['dob'] = pd.to\_datetime(df['dob'], errors='coerce')

# Placeholder removal

keywords = ['BABY', 'INFANT', 'VOID', 'UNKNOWN']

pattern = f"^({'|'.join(keywords)})"

placeholder\_mask = df['last\_name'].str.match(pattern, na=False)

df = df[~placeholder\_mask]

**Normalization Features**:

* Unicode normalization and diacritical mark removal
* Advanced name cleaning with suffix/prefix handling
* Geographic data standardization
* Placeholder record identification and removal
* Missing value imputation strategies

**2. Database Functions**

The system uses PostgreSQL functions for performance-critical operations:

**Utility Functions**:

* edit\_distance():Levenshtein distance calculation
* age\_difference(): Normalized age difference computation
* soundex(): Phonetic matching for names

**Blocking Functions**:

* create\_blocking\_keys(): Multi-strategy blocking key generation
* create\_blocking\_keys\_test(): Parallel worker version

**Comparison Functions**:

* compare\_records\_optimized(): Memory-efficient comparison with blocking
* compare\_records\_exhaustive(): Complete pairwise comparison

**Database Schema**

**Core Tables**

-- Records table (dynamic schema name for multi-tenancy)

CREATE TABLE records (

id TEXT PRIMARY KEY,

last\_name TEXT,

middle\_name TEXT,

ssn TEXT,

sex TEXT,

dob DATE,

phone TEXT,

zip TEXT,

city TEXT,

state TEXT,

address TEXT

);

-- Labels table (training pipeline only)

CREATE TABLE labels (

from\_id TEXT,

to\_id TEXT,

relationship TEXT

);

-- Processed records (comparison results)

CREATE TABLE processed\_records (

from\_id TEXT,

to\_id TEXT,

edit\_dist\_ln FLOAT,

edit\_dist\_phone\_num FLOAT,

edit\_dist\_mn FLOAT,

edit\_dist\_zip FLOAT,

edit\_dist\_city FLOAT,

age\_diff FLOAT,

sex\_diff INTEGER,

ssn\_match INTEGER,

state\_match INTEGER,

edit\_dist\_mail\_address FLOAT,

record1\_sex INTEGER,

record2\_sex INTEGER,

record1\_agecategory INTEGER,

record2\_agecategory INTEGER,

PRIMARY KEY (from\_id, to\_id)

);

**Blocking Infrastructure**

-- Blocking keys for efficient comparison

CREATE TABLE record\_blocks (

id TEXT,

block\_key TEXT,

sort\_key TEXT

);

-- Block size statistics

CREATE TABLE block\_sizes (

block\_key TEXT,

block\_size INTEGER

);