Cost Prediction App

Maintenance Guide

Mohamed Ashour

APC Mastery Path

Table of Contents

1 INTRODUCTION

1.1 Purpose of This Guide

This maintenance user guide provides comprehensive instructions for maintaining the Project Estimation ML App. It covers routine maintenance tasks, troubleshooting procedures, and best practices to ensure optimal performance and reliability.

1.2 Intended Audience

- System administrators
- IT support personnel
- Construction project managers
- Data analysts and ML practitioners
- End users responsible for system maintenance

1.3 Maintenance Philosophy

Regular maintenance ensures:

Optimal Performance: Consistent prediction accuracy and response times

Data Integrity: Clean, reliable datasets for training and prediction

System Reliability: Minimal downtime and error rates

Security: Protection of sensitive project data

Scalability: Ability to handle growing datasets and user demands

2 SYSTEM REQUIREMENTS

2.1 Minimum System Requirements

Operating System: Windows 10/11, macOS 10.14+, Linux (Ubuntu 18.04+)
Python Version: 3.7 or higher
RAM: 8GB minimum, 16GB recommended
Storage: 10GB free space minimum
CPU: Dual-core processor, quad-core recommended

2.2 Dependencies Overview

```
# Core ML Dependencies
pandas>=1.5.0          # Data manipulation
numpy>=1.21.0          # Numerical computing
scikit-learn>=1.1.0    # Machine learning algorithms
xgboost>=1.6.0         # Gradient boosting
scipy>=1.8.0           # Statistical functions

# Visualization Dependencies
matplotlib>=3.5.0      # Plotting library
seaborn>=0.11.0        # Statistical visualization

# File Format Support
openpyxl>=3.0.0        # Excel .xlsx support
pyxlsb>=1.0.0          # Excel .xlsb support
xlrd>=2.0.0            # Legacy .xls support

# Performance Enhancement
joblib>=1.1.0          # Parallel processing
```

2.3 Environment Validation Script

```
# Use this script to validate your environment
python -c "
import sys
print(f'Python version: {sys.version}')
import pandas, numpy, sklearn, xgboost, matplotlib
print('All core dependencies available')
"
```

3 INSTALLATION AND SETUP

3.1 Fresh Installation Process

3.1.1 Step 1: Environment Preparation

```
# Create virtual environment (recommended)
python -m venv ml_estimation_env

# Activate environment
# Windows:
ml_estimation_env\Scripts\activate
# macOS/Linux:
source ml_estimation_env/bin/activate
```

3.1.2 Step 2: Automated Installation

```
# Method 1: Use installation script
python install.py

# Method 2: Manual dependency installation
pip install -r Requirements.txt

# Method 3: Individual package installation
pip install pandas numpy matplotlib seaborn scikit-learn xgboost scipy openpyxl pyxlsb xlrd joblib
```

3.1.3 Step 3: Installation Verification

```
# Run the application launcher with checks
python run_app.py
```

3.2 Directory Structure Setup

```
Project-Estimation-ML-App/
├── project_estimation_app.py    # Main application
├── run_app.py                   # Launcher with validation
├── config.py                    # Configuration settings
├── batch_processor.py           # Batch processing utilities
├── data_validator.py            # Data validation
├── model_evaluator.py           # Model evaluation
├── debug_prediction.py          # Debug utilities
├── install.py                   # Installation script
├── Requirements.txt             # Dependencies
├── EULA.txt                     # End-user license
├── README.md                    # Documentation
├── data/                        # Input data directory
├── models/                      # Saved models
├── outputs/                     # Prediction outputs
├── logs/                        # Application logs
└── exports/                     # Export files
```

4 ROUTINE MAINTENANCE TASKS

4.1 Daily Maintenance Checklist

4.1.1 Application Health Check

```
# 1. Verify application starts correctly
python run_app.py

# 2. Check log files for errors
tail -f logs/app_launch_*.log

# 3. Verify disk space
df -h  # Linux/macOS
dir    # Windows
```

4.1.2 Performance Monitoring

- Monitor memory usage during large dataset processing
- Check CPU utilization during model training
- Verify response times for predictions

4.2 Weekly Maintenance Tasks

4.2.1 Log File Management

```
# Script to rotate and archive logs
import os
from datetime import datetime, timedelta

def archive_old_logs():
    log_dir = 'logs'
    archive_date = datetime.now() - timedelta(days=7)

    for file in os.listdir(log_dir):
        if file.endswith('.log'):
            file_path = os.path.join(log_dir, file)
            mod_time = datetime.fromtimestamp(os.path.getmtime(file_path))

            if mod_time < archive_date:
                # Archive or delete old logs
                archive_path = f'logs/archive/{file}'
                os.makedirs('logs/archive', exist_ok=True)
                os.rename(file_path, archive_path)
                print(f'Archived: {file}')
```

4.2.2 Data Quality Assessment

- Review prediction accuracy metrics
- Check for data drift in new datasets
- Validate model performance against benchmarks

4.3 Monthly Maintenance Tasks

4.3.1 Model Performance Review

```
# Model performance tracking script
import pandas as pd
import numpy as np
from datetime import datetime

def generate_performance_report():
    """Generate monthly model performance report"""
```

```
# Load recent prediction results
results = pd.read_csv('outputs/recent_predictions.csv')

# Calculate performance metrics
metrics = {
    'accuracy': calculate_accuracy(results),
    'prediction_count': len(results),
    'avg_confidence': results['confidence'].mean(),
    'processing_time': results['processing_time'].mean()
}

# Generate report
report = f"""
MONTHLY PERFORMANCE REPORT
Date: {datetime.now().strftime('%Y-%m-%d')}

Key Metrics:
- Prediction Accuracy: {metrics['accuracy']:.2%}
- Total Predictions: {metrics['prediction_count']}
- Average Confidence: {metrics['avg_confidence']:.2%}
- Average Processing Time: {metrics['processing_time']:.2f}s
"""

return report
```

5 DATA MANAGEMENT

5.1 Data Quality Maintenance

5.1.1 Automated Data Validation

```
# Use the built-in data validator
from data_validator import DataValidator

def validate_new_data(file_path):
    """Validate new datasets before processing"""
    validator = DataValidator()
    df = pd.read_csv(file_path)

    validation_results = validator.validate_dataset(df)

    if not validation_results['is_valid']:
        print("Data validation failed:")
        for error in validation_results['errors']:
            print(f"- {error}")
        return False

    print("Data validation passed")
    return True
```

5.1.2 Data Cleaning Procedures

```
def clean_project_data(df):
    """Standard data cleaning procedure"""

    # 1. Remove duplicates
    df = df.drop_duplicates()

    # 2. Handle missing values
    numeric_columns = df.select_dtypes(include=[np.number]).columns
    df[numeric_columns] = df[numeric_columns].fillna(df[numeric_columns].median())

    # 3. Remove outliers using IQR method
    for col in numeric_columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]

    # 4. Validate target variable
    if 'Estimate_at_Completion' in df.columns:
        df = df[df['Estimate_at_Completion'] > 0]

    return df
```

5.2 Data Backup Strategy

5.2.1 Automated Backup Script

```
import shutil
from datetime import datetime

def backup_data():
    """Create timestamped backup of data directory"""

    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_path = f'backups/data_backup_{timestamp}'

    # Copy data directory
    shutil.copytree('data', backup_path)
    print(f'Data backed up to: {backup_path}')
```

```
# Cleanup old backups (keep last 10)
cleanup_old_backups('backups', max_backups=10)
```

6 MODEL MAINTENANCE

6.1 Model Retraining Schedule

6.1.1 Automatic Retraining Trigger

```python
def check_retraining_needed():
    """Check if models need retraining"""

    # Criteria for retraining:
    # 1. Performance degradation
    # 2. New data availability
    # 3. Scheduled retraining interval

    last_training = get_last_training_date()
    days_since_training = (datetime.now() - last_training).days

    performance_metrics = get_recent_performance()
    accuracy_threshold = 0.85  # Minimum acceptable accuracy

    needs_retraining = (
        days_since_training > 30 or  # Monthly retraining
        performance_metrics['accuracy'] < accuracy_threshold or
        new_data_available()
    )

    return needs_retraining

def retrain_models():
    """Retrain all models with latest data"""

    print("Starting model retraining...")

    # 1. Load and prepare data
    df = load_latest_training_data()
    df = clean_project_data(df)

    # 2. Retrain models
    models = ['LinearRegression', 'RandomForest', 'XGBoost']
    for model_name in models:
        print(f"Retraining {model_name}...")
        model = train_model(df, model_name)
        save_model(model, model_name)

    # 3. Evaluate new models
    evaluate_all_models()

    print("Model retraining completed")
```

6.2 Model Performance Monitoring

6.2.1 Performance Tracking System

```python
class ModelPerformanceTracker:
    """Track model performance over time"""

    def __init__(self):
        self.performance_log = 'logs/model_performance.csv'

    def log_prediction(self, model_name, actual, predicted, features):
        """Log individual prediction for tracking"""

        entry = {
            'timestamp': datetime.now().isoformat(),
            'model': model_name,
            'actual': actual,
            'predicted': predicted,
            'error': abs(actual - predicted),
            'relative_error': abs(actual - predicted) / actual if actual != 0 else 0
        }
```

```
    # Append to performance log
    df = pd.DataFrame([entry])
    df.to_csv(self.performance_log, mode='a', header=False, index=False)

def generate_performance_report(self, days=30):
    """Generate performance report for specified period"""

    df = pd.read_csv(self.performance_log)
    df['timestamp'] = pd.to_datetime(df['timestamp'])

    # Filter recent data
    cutoff_date = datetime.now() - timedelta(days=days)
    recent_data = df[df['timestamp'] >= cutoff_date]

    # Calculate metrics by model
    metrics = recent_data.groupby('model').agg({
        'error': ['mean', 'std'],
        'relative_error': ['mean', 'std']
    }).round(4)

    return metrics
```

7 PERFORMANCE MONITORING

7.1 System Performance Metrics

7.1.1 Memory Usage Monitoring

```
import psutil
import matplotlib.pyplot as plt

def monitor_memory_usage():
    """Monitor and log memory usage during processing"""

    process = psutil.Process()
    memory_usage = []

    def log_memory():
        memory_info = process.memory_info()
        memory_usage.append({
            'timestamp': datetime.now(),
            'rss': memory_info.rss / 1024 / 1024,  # MB
            'vms': memory_info.vms / 1024 / 1024   # MB
        })

    return log_memory
```

7.1.2 Processing Time Analysis

```
import time
from functools import wraps

def measure_execution_time(func):
    """Decorator to measure function execution time"""

    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()

        execution_time = end_time - start_time
        print(f"{func.__name__} executed in {execution_time:.2f} seconds")

        # Log to performance file
        log_performance_metric(func.__name__, execution_time)

        return result

    return wrapper

# Usage example:
@measure_execution_time
def train_models(data):
    # Model training code here
    pass
```

7.2 Application Health Checks

7.2.1 Automated Health Check Script

```
def perform_health_check():
    """Comprehensive application health check"""

    health_report = {
        'timestamp': datetime.now().isoformat(),
        'status': 'HEALTHY',
        'issues': []
    }

    # Check 1: Dependencies
    try:
```

```
        import pandas, numpy, sklearn, xgboost
        health_report['dependencies'] = 'OK'
    except ImportError as e:
        health_report['dependencies'] = f'FAILED: {e}'
        health_report['issues'].append('Missing dependencies')
        health_report['status'] = 'UNHEALTHY'

    # Check 2: File system
    required_dirs = ['data', 'models', 'outputs', 'logs']
    for dir_name in required_dirs:
        if not os.path.exists(dir_name):
            health_report['issues'].append(f'Missing directory: {dir_name}')
            health_report['status'] = 'UNHEALTHY'

    # Check 3: Disk space
    disk_usage = psutil.disk_usage('.')
    free_space_gb = disk_usage.free / (1024**3)
    if free_space_gb < 5:  # Less than 5GB free
        health_report['issues'].append('Low disk space')
        health_report['status'] = 'WARNING'

    # Check 4: Memory
    memory = psutil.virtual_memory()
    if memory.percent > 90:  # More than 90% memory used
        health_report['issues'].append('High memory usage')
        health_report['status'] = 'WARNING'

    return health_report
```

8 TROUBLESHOOTING

8.1 Common Issues and Solutions

8.1.1 Issue 1: Import Errors

Error: ModuleNotFoundError: No module named 'pandas'
Solution:
1. Activate virtual environment: source ml_estimation_env/bin/activate
2. Install dependencies: pip install -r Requirements.txt
3. Verify installation: python -c "import pandas; print('OK')"

8.1.2 Issue 2: Memory Errors with Large Datasets

Error: MemoryError during data loading
Solutions:
1. Use batch processing:
   from batch_processor import BatchProcessor
   processor = BatchProcessor(chunk_size=1000)

2. Increase virtual memory (swap space)
3. Process data in smaller chunks
4. Use data sampling for initial analysis

8.1.3 Issue 3: GUI Not Starting

Error: tkinter.TclError
Solutions:
1. Check tkinter installation:
   python -c "import tkinter; print('OK')"

2. For Linux, install tkinter:
   sudo apt-get install python3-tk

3. For macOS with Homebrew:
   brew install python-tk

8.2 Debug Mode Usage

8.2.1 Running Debug Scripts

```
# Use debug_prediction.py for testing without GUI
python debug_prediction.py

# Enable verbose logging
import logging
logging.basicConfig(level=logging.DEBUG)

# Test specific components
from data_validator import DataValidator
validator = DataValidator()
# Test validation with sample data
```

8.3 Error Log Analysis

8.3.1 Log Analysis Script

```
def analyze_error_logs():
    """Analyze recent error logs for patterns"""

    import re
    from collections import Counter

    error_patterns = []

    # Read recent log files
    for log_file in glob.glob('logs/*.log'):
```

```
    with open(log_file, 'r') as f:
        for line in f:
            if 'ERROR' in line or 'CRITICAL' in line:
                error_patterns.append(line.strip())

# Count error types
error_types = Counter()
for error in error_patterns:
    if 'MemoryError' in error:
        error_types['Memory Issues'] += 1
    elif 'ImportError' in error:
        error_types['Import Issues'] += 1
    elif 'FileNotFoundError' in error:
        error_types['File Issues'] += 1
    else:
        error_types['Other'] += 1

return error_types
```

9 BACKUP AND RECOVERY

9.1 Backup Strategy

9.1.1 Complete System Backup

```
def create_full_backup():
    """Create complete system backup"""

    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_dir = f'backups/full_backup_{timestamp}'

    # Backup critical directories
    directories_to_backup = [
        'data',
        'models',
        'outputs',
        'logs',
        'exports'
    ]

    os.makedirs(backup_dir, exist_ok=True)

    for directory in directories_to_backup:
        if os.path.exists(directory):
            shutil.copytree(
                directory,
                os.path.join(backup_dir, directory)
            )
            print(f'Backed up: {directory}')

    # Backup configuration files
    config_files = [
        'config.py',
        'Requirements.txt',
        'EULA.txt'
    ]

    for file in config_files:
        if os.path.exists(file):
            shutil.copy2(file, backup_dir)

    print(f'Full backup created: {backup_dir}')
    return backup_dir
```

9.1.2 Incremental Backup System

```
def create_incremental_backup():
    """Create incremental backup of changed files"""

    last_backup_time = get_last_backup_time()
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_dir = f'backups/incremental_{timestamp}'

    os.makedirs(backup_dir, exist_ok=True)

    # Find files modified since last backup
    for root, dirs, files in os.walk('.'):
        for file in files:
            file_path = os.path.join(root, file)
            mod_time = datetime.fromtimestamp(os.path.getmtime(file_path))

            if mod_time > last_backup_time:
                # Copy modified file maintaining directory structure
                rel_path = os.path.relpath(file_path, '.')
                backup_path = os.path.join(backup_dir, rel_path)
                os.makedirs(os.path.dirname(backup_path), exist_ok=True)
                shutil.copy2(file_path, backup_path)
                print(f'Backed up modified file: {rel_path}')
```

9.2 Recovery Procedures

9.2.1 Data Recovery Script

```
def recover_from_backup(backup_path):
    """Recover system from backup"""

    if not os.path.exists(backup_path):
        raise FileNotFoundError(f"Backup not found: {backup_path}")

    print(f"Starting recovery from: {backup_path}")

    # Create recovery point of current state
    current_backup = create_full_backup()
    print(f"Current state backed up to: {current_backup}")

    # Restore from backup
    backup_contents = os.listdir(backup_path)

    for item in backup_contents:
        source_path = os.path.join(backup_path, item)

        if os.path.isdir(source_path):
            # Remove existing directory and restore
            if os.path.exists(item):
                shutil.rmtree(item)
            shutil.copytree(source_path, item)
            print(f"Restored directory: {item}")

        else:
            # Restore file
            shutil.copy2(source_path, item)
            print(f"Restored file: {item}")

    print("Recovery completed successfully")
```

10 SECURITY MAINTENANCE

10.1 Data Security Practices

10.1.1 Sensitive Data Handling

```
def sanitize_data_for_logging(data_dict):
    """Remove sensitive information from data before logging"""

    sensitive_fields = [
        'client_name',
        'project_address',
        'contact_info',
        'financial_details'
    ]

    sanitized = data_dict.copy()
    for field in sensitive_fields:
        if field in sanitized:
            sanitized[field] = '[REDACTED]'

    return sanitized
```

10.1.1.1 Access Control Verification

```
def verify_file_permissions():
    """Check and set appropriate file permissions"""

    # Set restrictive permissions on sensitive files
    sensitive_files = [
        'config.py',
        'data/',
        'models/',
        'logs/'
    ]

    for item in sensitive_files:
        if os.path.exists(item):
            # Set read/write for owner only
            os.chmod(item, 0o600)
            print(f"Updated permissions for: {item}")
```

10.2 Audit Trail Management

10.2.1 Activity Logging

```
class ActivityLogger:
    """Log user activities for audit trail"""

    def __init__(self):
        self.audit_log = 'logs/audit_trail.log'
        self.setup_logging()

    def setup_logging(self):
        """Setup audit logging configuration"""
        audit_logger = logging.getLogger('audit')
        handler = logging.FileHandler(self.audit_log)
        formatter = logging.Formatter(
            '%(asctime)s - AUDIT - %(message)s'
        )
        handler.setFormatter(formatter)
        audit_logger.addHandler(handler)
        audit_logger.setLevel(logging.INFO)
        self.logger = audit_logger

    def log_data_access(self, file_path, action):
        """Log data access events"""
        self.logger.info(f"Data {action}: {file_path}")

    def log_model_operation(self, operation, model_name):
```

```
    """Log model operations"""
    self.logger.info(f"Model {operation}: {model_name}")

def log_prediction_request(self, input_data, output):
    """Log prediction requests"""
    sanitized_input = sanitize_data_for_logging(input_data)
    self.logger.info(f"Prediction: Input={sanitized_input}, Output={output}")
```

11 UPDATES AND UPGRADES

11.1 Dependency Updates

11.1.1 Safe Update Process

```
def update_dependencies():
    """Safely update Python dependencies"""

    # 1. Create backup of current environment
    backup_requirements()

    # 2. Check for updates
    outdated_packages = check_outdated_packages()

    if outdated_packages:
        print(f"Found {len(outdated_packages)} outdated packages:")
        for package in outdated_packages:
            print(f"- {package}")

        # 3. Update in test environment first
        test_updates_in_virtual_env(outdated_packages)

        # 4. Apply updates if tests pass
        apply_updates(outdated_packages)

    else:
        print("All packages are up to date")

def backup_requirements():
    """Backup current requirements"""
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    backup_file = f'backups/requirements_backup_{timestamp}.txt'

    os.system(f'pip freeze > {backup_file}')
    print(f"Requirements backed up to: {backup_file}")
```

11.211.2 Application Updates

11.2.1 Version Control Integration

```
def check_for_updates():
    """Check for application updates"""

    current_version = "1.2.2"

    # In a real implementation, this would check a remote repository
    # or update server for newer versions

    print(f"Current version: {current_version}")
    print("Checking for updates...")

    # Placeholder for update checking logic
    # This would typically involve:
    # 1. Checking remote repository tags
    # 2. Comparing version numbers
    # 3. Downloading and applying updates

    return None  # No updates available
```

1212. LOG MANAGEMENT

12.112.1 Log Rotation Strategy

12.1.1.1 Automated Log Rotation

```
import logging.handlers

def setup_rotating_logs():
    """Setup log rotation to prevent disk space issues"""

    # Setup rotating file handler
    log_file = 'logs/application.log'
    handler = logging.handlers.RotatingFileHandler(
        log_file,
        maxBytes=10*1024*1024,  # 10MB per file
        backupCount=5           # Keep 5 backup files
    )

    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )
    handler.setFormatter(formatter)

    # Apply to root logger
    logger = logging.getLogger()
    logger.addHandler(handler)
    logger.setLevel(logging.INFO)

    return logger
```

12.212.2 Log Analysis Tools

12.2.1.1 Log Analysis Dashboard

```
def generate_log_analysis_report():
    """Generate comprehensive log analysis report"""

    report = {
        'timestamp': datetime.now().isoformat(),
        'log_files_analyzed': [],
        'error_summary': {},
        'performance_metrics': {},
        'recommendations': []
    }

    # Analyze all log files
    for log_file in glob.glob('logs/*.log'):
        analysis = analyze_single_log_file(log_file)
        report['log_files_analyzed'].append(analysis)

    # Generate recommendations
    if report['error_summary'].get('MemoryError', 0) > 5:
        report['recommendations'].append(
            "Consider increasing system memory or implementing data chunking"
        )

    if report['performance_metrics'].get('avg_processing_time', 0) > 60:
        report['recommendations'].append(
            "Processing times are high. Consider model optimization or hardware upgrade"
        )

    return report

def analyze_single_log_file(log_file):
    """Analyze individual log file"""

    analysis = {
        'file': log_file,
        'size_mb': os.path.getsize(log_file) / (1024*1024),
        'line_count': 0,
        'error_count': 0,
```

```
    'warning_count': 0,
    'last_modified': datetime.fromtimestamp(
        os.path.getmtime(log_file)
    ).isoformat()
}

with open(log_file, 'r') as f:
    for line in f:
        analysis['line_count'] += 1
        if 'ERROR' in line:
            analysis['error_count'] += 1
        elif 'WARNING' in line:
            analysis['warning_count'] += 1

return analysis
```

13 CONFIGURATION MANAGEMENT

13.1 Configuration Backup and Versioning

13.1.1 Configuration Management System

```python
class ConfigurationManager:
    """Manage application configuration versions"""

    def __init__(self):
        self.config_file = 'config.py'
        self.backup_dir = 'backups/config_versions'
        os.makedirs(self.backup_dir, exist_ok=True)

    def backup_current_config(self):
        """Backup current configuration"""
        timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
        backup_file = os.path.join(
            self.backup_dir,
            f'config_backup_{timestamp}.py'
        )

        shutil.copy2(self.config_file, backup_file)
        print(f"Configuration backed up to: {backup_file}")
        return backup_file

    def restore_config(self, backup_file):
        """Restore configuration from backup"""
        if not os.path.exists(backup_file):
            raise FileNotFoundError(f"Backup file not found: {backup_file}")

        # Backup current config before restore
        self.backup_current_config()

        # Restore from backup
        shutil.copy2(backup_file, self.config_file)
        print(f"Configuration restored from: {backup_file}")

    def list_config_versions(self):
        """List available configuration versions"""
        versions = []
        for file in os.listdir(self.backup_dir):
            if file.startswith('config_backup_'):
                versions.append({
                    'file': file,
                    'timestamp': file.replace('config_backup_', '').replace('.py', ''),
                    'path': os.path.join(self.backup_dir, file)
                })

        return sorted(versions, key=lambda x: x['timestamp'], reverse=True)
```

13.2 Environment-Specific Configurations

13.2.1 Configuration Templates

```python
# Development configuration
DEVELOPMENT_CONFIG = {
    'APP_NAME': "Project Estimation ML App (Dev)",
    'LOG_LEVEL': 'DEBUG',
    'MAX_FILE_SIZE_MB': 100,
    'CHUNK_SIZE': 1000
}

# Production configuration
PRODUCTION_CONFIG = {
    'APP_NAME': "Project Estimation ML App",
    'LOG_LEVEL': 'INFO',
    'MAX_FILE_SIZE_MB': 500,
    'CHUNK_SIZE': 5000
}
```

```python
def apply_environment_config(environment='production'):
    """Apply environment-specific configuration"""

    config_map = {
        'development': DEVELOPMENT_CONFIG,
        'production': PRODUCTION_CONFIG
    }

    if environment not in config_map:
        raise ValueError(f"Unknown environment: {environment}")

    config = config_map[environment]

    # Update config.py file
    with open('config.py', 'r') as f:
        content = f.read()

    for key, value in config.items():
        # Replace configuration values
        pattern = f'{key} = .*'
        replacement = f'{key} = {repr(value)}'
        content = re.sub(pattern, replacement, content)

    with open('config.py', 'w') as f:
        f.write(content)

    print(f"Applied {environment} configuration")
```

14 APPENDICES

14.1 Appendix A: Command Reference

14.1.1 Essential Commands

```
# Application Management
python run_app.py                # Start application with checks
python install.py                # Install dependencies
python debug_prediction.py       # Run debug mode

# Dependency Management
pip install -r Requirements.txt  # Install all dependencies
pip freeze > Requirements.txt    # Save current dependencies
pip list --outdated              # Check for updates

# System Monitoring
python -c "import psutil; print(f'Memory: {psutil.virtual_memory().percent}%')"
python -c "import psutil; print(f'Disk: {psutil.disk_usage(\".\").percent}%')"

# Log Management
tail -f logs/app_launch_*.log    # Monitor latest logs
find logs -name "*.log" -mtime +7  # Find logs older than 7 days
```

14.2 Appendix B: Performance Benchmarks

14.2.1 Expected Performance Metrics

Dataset Size	Loading Time	Training Time	Prediction Time
< 1,000 rows	< 1 second	< 5 seconds	< 0.1 seconds
1,000-10,000	< 5 seconds	< 30 seconds	< 0.5 seconds
10,000-50,000	< 30 seconds	< 2 minutes	< 2 seconds
50,000+	Use batch	< 10 minutes	< 10 seconds

14.2.2 Memory Usage Guidelines

Dataset Size	Memory Usage	Recommended RAM
< 1,000 rows	< 100 MB	4 GB
1,000-10,000	< 500 MB	8 GB
10,000-50,000	< 2 GB	16 GB
50,000+	Variable	32 GB+

14.3 Appendix C: Error Code Reference

14.3.1 Common Error Codes and Solutions

ML_001: Data loading failed
- Check file format and permissions
- Verify file is not corrupted

ML_002: Insufficient training data
- Ensure minimum 10 samples
- Check for valid target values

ML_003: Memory allocation error
- Reduce batch size in config
- Use chunked processing

ML_004: Model training failed
- Check feature data types
- Verify target variable format

ML_005: Prediction service unavailable
- Restart application
- Check model files exist

14.4 Appendix D: Contact Information

Technical Support: - Website: www.apcmasterypath.co.uk - LinkedIn: https://www.linkedin.com/in/mohamed-ashour-0727/ - YouTube: APC Mastery Path (https://www.youtube.com/@APCMasteryPath)

Emergency Procedures:

1. Check system health: `python -c "from run_app import perform_health_check; print(perform_health_check())"`
2. Create emergency backup: `python -c "from maintenance import create_full_backup; create_full_backup()"`
3. Review recent logs: `tail -50 logs/app_launch_*.log` 4. Contact support with error details and log files

---

Disclaimer

This maintenance guide should be reviewed and updated quarterly to ensure accuracy and completeness. Last updated: September 2025