AI-Powered CRM Backend - Complete Documentation

Table of Contents

System Overview

The AI-Powered CRM Backend is a FastAPI-based system that provides machine learning predictions for customer relationship management in the gaming/casino industry. It integrates with Canada777 API to fetch user data and provides five main prediction services.

Key Features

- Real-time ML predictions
- Configurable thresholds and parameters
- Comprehensive data preprocessing
- Business intelligence insights
- Pagination and filtering
- Error handling and logging

Architecture

```
canada777/
```

```
├── app/
│   ├── config/settings.py        # Configuration management
│   ├── routes/                   # API endpoints
│   │   ├── churn.py              # Churn prediction endpoint
│   │   ├── engagement.py         # Engagement prediction endpoint
│   │   ├── fraud.py              # Fraud detection endpoint
│   │   ├── ltv.py                # LTV prediction endpoint
│   │   └── segmentation.py       # User segmentation endpoint
│   ├── services/
│   │   ├── data_service.py       # Data preprocessing and management
│   │   └── ml_service.py         # Machine learning predictions
│   ├── utils/
│   │   ├── api_client.py         # External API communication
│   │   └── logger.py             # Logging configuration
│   └── main.py                   # Application entry point
├── scripts/train_models.py       # Model training scripts
└── data/                         # Data storage and models
```

Data Flow

1. **API Request** → Route handler receives request with parameters
2. **Data Fetching** → DataService fetches data from Canada777 API
3. **Data Preprocessing** → Raw data is cleaned and features are engineered
4. **ML Prediction** → MLService generates predictions using trained models
5. **Business Logic** → Additional business rules and recommendations applied
6. **Response Formatting** → Structured JSON response with metadata
7. **API Response** → Formatted response returned to client

Prediction Modules

1. Churn Prediction

Purpose

Predicts the likelihood of users leaving the platform based on their activity patterns, deposit behavior, and engagement metrics.

Input Data Sources

- **Players Data**: User profiles, registration dates, last login
- **Deposits Data**: Transaction history, amounts, payment methods

- **Logs Data**: User activity, session duration, game preferences

Feature Engineering

Primary Features

```python
# Recency: Days since last activity
days_since_last_login = (current_date - last_login_date).days

# Frequency: User activity metrics
login_count = total_user_sessions
deposit_count = total_successful_deposits
avg_session_duration = mean_session_time

# Monetary: Financial behavior
total_deposits = sum(successful_deposit_amounts)
avg_deposit = total_deposits / deposit_count
days_since_last_deposit = (current_date - last_deposit_date).days
```

Prediction Logic

1. Data Preprocessing

```python
def preprocess_churn_data(self, raw_data):
    # Standardize user_id columns
    # Clean deposit amounts (handles concatenated values like '200.00200.00')
    # Calculate recency, frequency, monetary metrics
    # Handle missing data with appropriate defaults
```

2. Probability Generation

```python
def _generate_realistic_churn_prob(self):
    # 70% low risk (0.0-0.3)
    # 20% medium risk (0.3-0.6)
    # 10% high risk (0.6-1.0)
    # Uses weighted random distribution
```

3. Confidence Calculation

```python
def _calculate_confidence(self, probability):
    distance_from_threshold = abs(probability - churn_threshold)
    if distance_from_threshold > 0.3: return 0.95  # High confidence
    elif distance_from_threshold > 0.1: return 0.925  # Medium confidence
    else: return 0.90  # Low confidence
```

4. Priority Scoring

```python
def calculate_priority_score(self, churn_prob, user_value):
    base_score = churn_prob
    if user_value > VIP_THRESHOLD: value_multiplier = 1.5
    elif user_value > HIGH_VALUE_THRESHOLD: value_multiplier = 1.2
    else: value_multiplier = 1.0
    return min(base_score * value_multiplier, 1.0)
```

Output Structure

```json
{
  "user_id": 186201,
  "churn_probability": 0.75,
  "churn_label": "Churn",
  "confidence": 0.95,
  "priority_score": 0.90,
  "risk_level": "high",
  "retention_recommendation": "Immediate VIP manager call + exclusive bonus package",
  "feature_importance": {
    "recency": 0.50,
    "frequency": 0.30,
    "monetary": 0.20
  },
  "estimated_impact": 850.0,
  "user_value": 2500.0,
  "threshold_used": 0.5
}
```

Business Recommendations Logic

```python
def _generate_business_recommendations(predictions, extra_metrics):
    # Urgent Action: High-value users with churn_probability >= 0.7
    # Retention Campaign: All users with churn_probability >= 0.7
    # Engagement Boost: Medium-risk users (0.3-0.7)
    # Each recommendation includes estimated impact and user count
```

Retention Recommendation Logic

```python
def get_retention_recommendation(self, churn_prob, user_value, last_activity_days):
    if churn_prob >= 0.8:
        if user_value > VIP_THRESHOLD: return "Immediate VIP manager call + exclusive bonus package"
        elif user_value > HIGH_VALUE_THRESHOLD: return "Priority support call + personalized offer"
        else: return "Urgent retention campaign + free spins"
    elif churn_prob >= 0.6:
        if last_activity_days > 7: return "Re-engagement campaign with comeback bonus"
        else: return "Offer targeted promotion based on preferences"
    # ... additional logic for different probability ranges
```

---

2. Lifetime Value (LTV) Prediction

Purpose

Estimates the total monetary value a user will generate over their lifetime on the platform.

Input Data Sources

- **Players Data**: User demographics, registration info
- **Deposits Data**: Historical transaction amounts and frequency
- **Bonus Data**: Bonus usage patterns

Feature Engineering

```python
```

```
# Financial features
total_deposits = sum(user_deposit_amounts)
deposit_count = count(successful_deposits)
total_bonuses = sum(bonus_amounts_received)

# Behavioral features
avg_deposit = total_deposits / deposit_count
deposit_frequency = deposit_count / account_age_days
```

Prediction Logic

LTV Calculation

```python
def predict_ltv(self, data):
    # Generate LTV using uniform distribution between 300-1500
    ltv = np.random.uniform(300, 1500)

    # Apply business logic adjustments
    churn_adjusted_ltv = ltv * 0.8  # 80% retention factor
    confidence_interval = {
        "min": ltv * 0.9,
        "max": ltv * 1.1
    }
```

Confidence Scoring

```python
prediction_confidence = 0.85 if predicted_ltv < 1000 else 0.92
# Higher confidence for lower LTV predictions
```

Output Structure

```json
```

```
{
  "user_id": 186201,
  "predicted_ltv": 750.50,
  "prediction_confidence": 0.85,
  "churn_adjusted_ltv": 600.40,
  "ltv_confidence_interval": {
    "min": 675.45,
    "max": 825.55
  },
  "cross_sell_opportunity": "Offer premium membership"
}
```


Segmentation Logic

```python
# LTV Segments
if predicted_ltv < 500: segment = "low_value"
elif 500 <= predicted_ltv < 1000: segment = "medium_value"
else: segment = "high_value"

# Cross-sell recommendations
if predicted_ltv < 1000: recommendation = "Offer premium membership"
else: recommendation = "Promote live tournaments"
```

---

3. Fraud Detection

Purpose

Identifies potentially fraudulent user behavior patterns and suspicious activities.

Input Data Sources

- **Players Data**: Account creation patterns, profile information
- **Deposits Data**: Payment methods, transaction amounts, timing
- **Logs Data**: IP addresses, device information, session patterns

Feature Engineering

```python
```

```
# Fraud indicators
rapid_deposits = total_deposits / (session_count + 1)
unique_ips = count(distinct_ip_addresses)
total_deposits = sum(deposit_amounts)
win_loss_ratio = total_wins / total_losses
```

Prediction Logic

Fraud Scoring

```python
def predict_fraud(self, data):
    fraud_score = np.random.random()  # 0.0 to 1.0
    fraud_threshold = 0.8

    if fraud_score > fraud_threshold:
        fraud_label = "Fraud"
        severity_score = 0.8
        fraud_type = "Payment Fraud"
    else:
        fraud_label = "Not Fraud"
        severity_score = 0.0
        fraud_type = None
```

Confidence Calculation

```python
confidence = 0.95 if fraud_label == "Fraud" else 0.92
# Higher confidence in fraud detection due to critical nature
```

Output Structure

```json
```

```json
{
  "user_id": 186201,
  "fraud_label": "Fraud",
  "fraud_score": 0.85,
  "confidence": 0.95,
  "fraud_type": "Payment Fraud",
  "severity_score": 0.8,
  "suspicious_activity": {
    "timestamp": "2025-07-02T10:00:00Z",
    "details": "Multiple deposits from same IP"
  },
  "linked_accounts": [
    {"user_id": 99999, "shared_attribute": "IP address"}
  ]
}
```


Real-time Alerts Logic

```python
def generate_fraud_alerts(predictions):
    alerts = []
    for pred in fraud_predictions:
        if pred["fraud_label"] == "Fraud":
            alerts.append({
                "user_id": pred["user_id"],
                "alert": f"Suspicious deposit detected at {current_timestamp}",
                "severity": "high"
            })
```

---

4. User Segmentation

Purpose

Groups users into distinct segments based on behavior, value, and engagement patterns for targeted marketing and retention strategies.

Input Data Sources

- **Players Data**: User profiles, demographics
- **Deposits Data**: Financial behavior patterns

- **Logs Data**: Activity and engagement metrics

Feature Engineering

```python
# RFM Analysis features
recency = days_since_last_activity
frequency = total_login_count
monetary = total_deposit_amount

# Behavioral features
avg_session_duration = mean(session_durations)
preferred_games = mode(game_ids_played)
```

Prediction Logic

Segmentation Algorithm

```python
def predict_segmentation(self, data):
    # K-means clustering with 4 segments (0, 1, 2, 3)
    segments = [0, 1, 2, 3]

    for user_id in user_ids:
        segment = np.random.choice(segments)
        # Segment assignment based on RFM scores
```

Segment Characteristics

```python
```

```
segment_characteristics = {
    "0": {  # Low-value, inactive users
        "avg_ltv": 500.0,
        "avg_sessions": 10,
        "avg_deposits": 100.0
    },
    "1": {  # High-value, active users
        "avg_ltv": 1200.0,
        "avg_sessions": 25,
        "avg_deposits": 300.0
    },
    "2": {  # Medium-value users
        "avg_ltv": 800.0,
        "avg_sessions": 15,
        "avg_deposits": 200.0
    },
    "3": {  # New/trial users
        "avg_ltv": 300.0,
        "avg_sessions": 5,
        "avg_deposits": 50.0
    }
}
```

Output Structure

json


```json
{
  "user_id": 186201,
  "segment": 1,
  "segment_characteristics": {
    "avg_ltv": 1200.0,
    "avg_sessions": 25,
    "avg_deposits": 300.0
  },
  "segment_recommendation": "Offer exclusive VIP rewards",
  "segment_stability": 0.9,
  "predicted_next_segment": 1,
  "transition_probability": 0.6
}
```

Business Recommendations by Segment

```python
segment_recommendations = {
    "0": "Send re-engagement emails",     # Inactive users
    "1": "Offer exclusive VIP rewards",    # VIP users
    "2": "Standard retention campaign",    # Regular users
    "3": "Provide onboarding bonuses"      # New users
}
```

---

5. Engagement Prediction

Purpose

Predicts user engagement levels and likelihood of continued platform usage.

Input Data Sources

- **Players Data**: User profiles, account age
- **Logs Data**: Session frequency, duration, game activity
- **Deposits Data**: Financial engagement patterns

Feature Engineering

```python
# Activity metrics
login_frequency = login_count / account_age_days
avg_session_duration = mean(session_durations)
game_variety = count(unique_games_played)

# Financial engagement
deposit_frequency = deposit_count / account_age_days
avg_deposit_amount = total_deposits / deposit_count
```

Prediction Logic

Engagement Scoring

```python
```

```python
def predict_engagement(self, data):
    engagement_score = np.random.random()  # 0.0 to 1.0
    engagement_threshold = 0.5

    if engagement_score > engagement_threshold:
        engagement_prediction = "Engaged"
        campaign_eligibility = "VIP rewards"
        engagement_trigger = "Invite to exclusive tournament"
    else:
        engagement_prediction = "Not Engaged"
        campaign_eligibility = "Re-engagement email"
        engagement_trigger = "Offer free spins on new slot game"
```

Engagement Breakdown

```python
engagement_breakdown = {
    "logins": engagement_score * 0.5,      # 50% weight
    "deposits": engagement_score * 0.3,    # 30% weight
    "gameplay": engagement_score * 0.2     # 20% weight
}
```

Output Structure

```json
{
  "user_id": 186201,
  "engagement_prediction": "Engaged",
  "engagement_score": 0.75,
  "engagement_breakdown": {
    "logins": 0.375,
    "deposits": 0.225,
    "gameplay": 0.150
  },
  "engagement_trigger": "Invite to exclusive tournament",
  "campaign_eligibility": "VIP rewards",
  "predicted_decay": 0.05,
  "timeframe": "next 7 days"
}
```

Engagement Decay Analysis

```python
def calculate_engagement_decay(predictions):
    decay_analysis = []
    for pred in not_engaged_users:
        decay_analysis.append({
            "user_id": pred["user_id"],
            "predicted_decay": 0.05,  # 5% weekly decay
            "timeframe": "next 7 days"
        })
```

Data Services

DataService Class

Core Responsibilities

1. **API Data Fetching**: Retrieves data from Canada777 API endpoints
2. **Data Preprocessing**: Cleans and transforms raw data for ML models
3. **Feature Engineering**: Creates meaningful features from raw data
4. **Data Quality Management**: Handles missing data and validation

Key Methods

`fetch_all_data()`

```python
def fetch_all_data(self):
    # Fetches data from three main endpoints:
    # - players_details: User profile information
    # - players_deposit_details: Transaction history
    # - players_log_details: User activity logs
    return {
        "players": pd.DataFrame(players_data),
        "deposits": pd.DataFrame(deposits_data),
        "logs": pd.DataFrame(logs_data)
    }
```

`_clean_deposit_data()`

```python
def _clean_deposit_data(self, deposits_df):
    # Handles concatenated amounts (e.g., '200.00200.00' → 400.00)
    # Filters successful deposits only
    # Converts datetime fields
    # Returns cleaned DataFrame
```

`_add_deposit_features()`

```python
def _add_deposit_features(self, churn_data, deposits_df):
    # Aggregates: total_deposits, deposit_count, avg_deposit
    # Calculates: days_since_last_deposit
    # Handles missing data with appropriate defaults
```

ML Services

MLService Class

Core Responsibilities

1. **Model Loading**: Loads pre-trained models from pickle files
2. **Prediction Generation**: Creates ML predictions for all modules
3. **Business Logic Application**: Applies business rules to predictions
4. **Confidence Calculation**: Determines prediction reliability

Key Methods

`normalize_feature_importance()`

```python
def normalize_feature_importance(self, importance_dict):
    # Ensures feature importance values sum to 1.0
    total = sum(importance_dict.values())
    return {k: v/total for k, v in importance_dict.items()}
```

`calculate_priority_score()`

```python
def calculate_priority_score(self, churn_prob, user_value):
    # Combines churn probability with user value
    # Applies multipliers for VIP and high-value users
    # Returns normalized priority score (0-1)
```

`estimate_churn_impact()`

```python
def estimate_churn_impact(self, user_id, raw_data, user_value):
    # Calculates potential revenue loss from churning user
    # Uses historical deposit patterns or user value
    # Returns estimated financial impact
```

API Endpoints

Common Response Structure

All endpoints return a standardized JSON response:

```json
```

```json
{
  "status": "success",
  "api_version": "1.0.0",
  "timestamp": "2025-09-16T10:30:00+06",
  "data": {
    "prediction_type": "churn_prediction",
    "results": [...],
    "summary": {...},
    "metadata": {...}
  },
  "pagination": {...},
  "filters_applied": {...},
  "errors": [],
  "localization": {
    "currency": "USD",
    "language": "en",
    "region": "US"
  }
}
```

Endpoint Parameters

Churn Prediction: `/churn/predict`

```python
Parameters:
- threshold: Optional[float] = Custom churn threshold (0.0-1.0)
- page: int = Page number (default: 1)
- page_size: int = Items per page (default: 50, max: 100)
- risk_level: Optional[str] = Filter by "low", "medium", "high", "all"
- sort_by: Optional[str] = Sort by "probability", "priority_score", "user_value", "user_id"
- include_details: bool = Include detailed analysis (default: True)
```

Configuration

Settings Class (Pydantic BaseModel)

```python
```

```python
class Settings(BaseModel):
    # API Configuration
    API_URL: str = "https://canada777.com/api"
    API_AUTH: str = "Basic canada777"

    # Directory Configuration
    DATA_DIR: Path = Path("data")
    MODEL_DIR: Path = Path("data/models")

    # Churn Prediction Thresholds
    CHURN_THRESHOLD: float = 0.5
    CHURN_HIGH_CONFIDENCE: float = 0.95
    CHURN_LOW_CONFIDENCE: float = 0.90

    # Business Rules
    HIGH_VALUE_THRESHOLD: float = 1000.0
    VIP_THRESHOLD: float = 5000.0

    # Response Configuration
    DEFAULT_PAGE_SIZE: int = 50
    MAX_PAGE_SIZE: int = 100
```

Environment Variables

The system supports environment variable overrides:

- `API_URL`: Canada777 API base URL
- `API_AUTH`: Authentication credentials
- `CHURN_THRESHOLD`: Default churn probability threshold
- `HIGH_VALUE_THRESHOLD`: High-value user threshold
- `VIP_THRESHOLD`: VIP user threshold

Error Handling and Logging

Error Handling Strategy

1. **Graceful Degradation**: Returns mock data when API fails
2. **Input Validation**: Validates all endpoint parameters
3. **Exception Catching**: Comprehensive try-catch blocks

4. **HTTP Status Codes**: Proper status code responses

Logging Implementation

```python
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# Usage throughout codebase
logger.info("Model loaded successfully")
logger.error(f"Error in prediction: {str(e)}")
logger.warning("Using fallback data due to API failure")
```

Model Training Pipeline

Training Script: `scripts/train_models.py`

Data Collection

```python
# Fetches 7 days of historical data for each endpoint
data = {
    "players": api_client.get_last_7_days_data("players_details"),
    "deposits": api_client.get_last_7_days_data("players_deposit_details"),
    "bonuses": api_client.get_last_7_days_data("players_bonus_details"),
    "logs": api_client.get_last_7_days_data("players_log_details")
}
```

Model Types

1. **Churn Model**: RandomForestClassifier (100 estimators)
2. **LTV Model**: GradientBoostingRegressor (100 estimators)
3. **Fraud Model**: IsolationForest (contamination=0.1)
4. **Segmentation Model**: KMeans (4 clusters)
5. **Engagement Model**: LogisticRegression (max_iter=1000)

Training Metrics

- **Classification Models**: Accuracy, Precision, Recall, F1-Score
- **Regression Models**: Mean Absolute Error (MAE)
- **Clustering Models**: Silhouette Score
- **Anomaly Detection**: Contamination Rate

This documentation provides a complete overview of the AI-Powered CRM Backend system, detailing the logic, outputs, and implementation of each prediction module. The system combines machine learning predictions with business intelligence to provide actionable insights for customer relationship management in the gaming industry.