**Quantitative ML Trading Framework** *© 2025 Lukas Svešnikovas. All rights reserved.*

**Introduction**

The cryptocurrency market is characterized by high volatility, liquidity fluctuations, and rapid price movements. Traditional technical analysis methods are often insufficient for accurately predicting short-term price action. In recent years, the application of artificial intelligence and machine learning (ML) in finance has gained significant traction, with the aim of detecting recurring market patterns and uncovering sustainable trading edges.

Major players in the industry — quantitative funds and crypto-focused hedge funds — extensively employ ML techniques. Each type of model addresses different challenges using unique statistical architectures, ranging from microstructure signal detection and short-term order-flow modeling to regime prediction and risk-management optimization.

The goal of this project is to design and implement a complete ML-based trading system for cryptocurrency signal forecasting and strategy backtesting. The system covers the entire data processing cycle: historical OHLCV and order-flow data collection, preprocessing, feature engineering, model training, CPU-based parallelization across all cores with RAM usage monitoring, backtesting, and performance evaluation on more than ten years of BTC/USDT (or other liquid cryptocurrency) 1-minute data.

The models applied include **XGBoost**, which focuses on microstructure-level signal prediction using **2D feature matrices** (per-candle features), as well as deep learning architectures such as **CNN** and **LSTM**, which leverage **3D inputs** (sequence length × features per candle) for regime detection and macro-micro structural learning. The system enables not only signal-accuracy evaluation but also full trading-performance analysis, including win rate, risk-reward ratio, balance trajectory, and maximum drawdown, with both commissions and leverage explicitly accounted for.

The entire project is implemented in **Python**, with **PostgreSQL/SQL** used for data storage and large-scale dataset handling.

**What the XGBoost Model Currently Includes**

*(the only ML model published so far from me)*

**Data & Labels**

* Label generation for multiple TP/SL/lookahead combinations (direction-aware: \_long / \_short).
* AEA gating (accumulation/distribution, spikes, RSI/ADX, OF-5/15, trend) to reduce label noise.
* Feature validation: dtype coercion, missing-column checks, entry price calculation, and time\_to\_hit.

**Training & Parallelization**

* Joblib **loky** parallel grid over (tp, sl, lookahead) with chunked training.
* Multiprocessing task parallelism controlled by n\_jobs, batch\_size.
* Global class encoder (encoder.joblib) shared across all runs.
* Model checkpointing (checkpoints/model\_part\*.joblib) with best-model selection.
* Per-chunk feature-importance extraction (“žmogystai”).

**Memory & Performance**

* RAM guards: estimate\_gb, system\_used\_ram\_gb, and chunked I/O to prevent memory overflow.
* Unified progress bars via **tqdm** and queue updater.
* Optional downsampling for computationally heavy tasks (e.g., SHAP).

**Evaluation & Reporting**

* Mastery roll-ups: TP/SL hits, win-rate, average time-to-hit, and long/short summaries.
* Black-box opinion: predict\_proba + confidence gates + gate\_long/short for per-label execution stats.
* SHAP explainability with noise filtering and class-level diagnostics.
* Best-run tracker: best\_blackbox\_opinion.txt and best\_metric\_\*.txt.

**Simulation & Risk**

* Stage-2 simulation with leverage and fees, including reason-coded exits (TP/SL/next signal).
* Position sizing capped by max\_used\_balance.
* Outputs: preds\_log\_\*.csv, simuliacijos\_outputas\_\*.csv, and feedback bank (boost factors: win = 1.2, loss = 0.8).

**Reproducibility & Infrastructure**

* Python 3.11+, dependencies: xgboost, scikit-learn, pandas, numpy, joblib, psutil, pympler, scipy, shap, matplotlib.
* Windows freeze\_support() for safe multiprocessing.
* Deterministic seeds, consistent feature lists, and a documented **How-to-Run** workflow.

**Files Produced**

* labeliai/\*.csv — generated labels
* checkpoints/\*.joblib — trained models
* mastery\_\*.csv — mastery roll-ups
* black\_box\_opinion\_\*.csv — confidence-gated execution stats
* preds\_log\_\*.csv, simuliacijos\_outputas\_\*.csv — predictions and simulated trades
* feedback\_bank\_\*.csv — per-bar feedback signals

**Pipeline Architecture**

**Stage 1 (Data Acquisition and Feature Engineering).**  
This stage needs to be executed only once, as it prepares the dataset on which the models will later be trained. Even if model training fails (e.g., due to an error), Stage 1 does not need to be repeated.

Data is collected using **CCXT** for OHLCV and **order-flow (OF)** archives. To accelerate the process, **parallelization across all CPU cores** is applied, mainly through Python’s multiprocessing with task parallelism (n\_workers, threads).

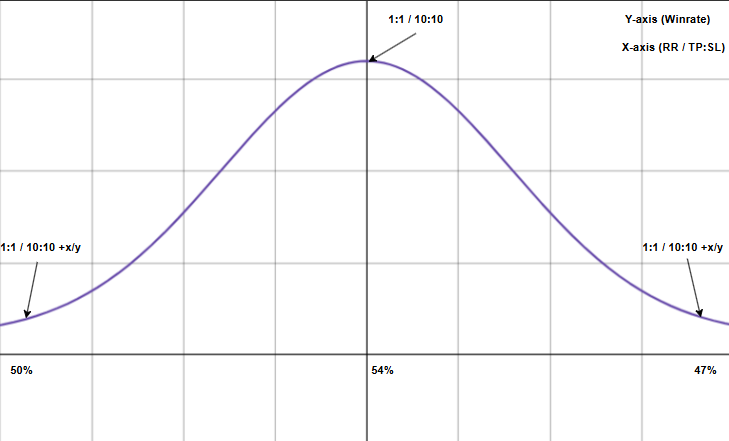
Because of the dataset’s size, the data is stored in a **PostgreSQL/SQL** database. (Contrary to a common misconception, CSV files do not have strict row limits; however, practical constraints such as Excel’s 1,048,576-row limit and file size/IO performance make databases a more robust solution.)

From the collected CCXT and OF data, **51 features** are currently engineered. Many of them are log-transformed to stabilize variance and improve recognition by **XGBoost**. Incorporating OF metrics (buy/sell counts, volumes, deltas, etc.) provides a significant edge compared to OHLCV-only datasets.

**Stage 2 (Machine Learning and Simulation).**

This stage encompasses model training, evaluation, and simulation. As of now, three models have been developed, each addressing different tasks; two are still under active development.

* **XGBoost**: used for microstructure confirmation. Current accuracy is ~54%. With a risk–reward ratio of 1:1, the breakeven threshold for profitability is ~57%. Without trading fees, the model would already be profitable, since it exceeds the 50% baseline required for 1:1 setups.
* Importantly, the model operates on **2D inputs** (per-candle feature vectors). For example, it uses the features of a single 1-minute candle to predict whether the next move is **up** or **down**.
* The decision threshold for classification is currently set to **±10% price movement** in either direction. As illustrated in the figure, these TP/SL values have been systematically tested, showing the relationship between win rate and RR (risk–reward).



The **2D XGBoost** module currently spans ~2.5k lines of Python (consolidated into a single training/simulation pipeline). The other two models are in active development.

**AI2D.py — Technical Documentation (Extended)**

This section provides a deeper, code-level specification of **AI2D.py**: its responsibilities, data flow, key functions, produced artifacts, and end-to-end execution steps.

* **Responsibilities:** label generation (TP/SL/lookahead), model training/checkpointing, mastery roll-ups, black-box opinion, and Stage-2 trade simulation with fees/leverage.
* **Data Flow:** load feature-rich 1m data → compute AEA gates → create labels → train XGBoost per combo → export artifacts → simulate trades.
* **Key Functions:** compute\_aea\_signals, train\_on\_file, generate\_mastery, black\_box\_opinion, run\_stage2.
* **Artifacts:** labeliai/\*.csv, checkpoints/\*.joblib, mastery\_\*.csv, black\_box\_opinion\_\*.csv, preds\_log\_\*.csv, simuliacijos\_outputas\_\*.csv.
* **How to Run:** prepare WITH\_BTC\_USDT\_1m\_part\*.csv, set TP/SL/lookahead grid and resource params, execute AI2D.py to generate labels, train models, and run simulation.

**1) Purpose and Scope**

**AI2D.py** implements Stage‑2/Stage‑1.5 of the trading pipeline:

* Generates supervised **labels** for multiple TP/SL/lookahead combos over 1‑minute BTC/USDT data enhanced with order‑flow features.
* Trains **XGBoost** classifiers per label chunk with checkpoints and early model selection.
* Produces **feature‑importance summaries** ("žmogystai") and **mastery** roll‑ups.
* Builds a **black\_box\_opinion** report that aggregates class‑wise execution decisions and win‑rates.
* Runs **Stage‑2 simulation** (signal execution gates + TP/SL PnL logic) to produce simuliacijos\_outputas\_\*.csv and per‑bar prediction logs + feedback.

**Input:** feature‑enriched CSVs (e.g., WITH\_BTC\_USDT\_1m\_part\*.csv) created by the earlier pipeline stages.

**Output (artifacts):**

* labeliai/… — per‑chunk label files.
* checkpoints/model\_part\*.joblib + encoder.joblib — model artifacts.
* mastery\_tp{…}\_sl{…}\_look{…}\_part{…}.csv — mastery roll‑ups.
* black\_box\_opinion\_tp{…}\_sl{…}\_look{…}\_part{…}.csv — confidence‑gated execution stats.
* preds\_log\_tp{…}\_sl{…}.csv, simuliacijos\_outputas\_tp{…}\_sl{…}.csv, feedback\_bank\_tp{…}\_sl{…}.csv — simulation & feedback products.

**2) High‑Level Data Flow**

1. **Load base data:** read OHLVC + order‑flow features (WITH\_BTC\_USDT\_1m\_part\*.csv).
2. **AEA filters:** compute regime/quality flags (accumulation/distribution, spikes, momentum, RSI/ADX, OF 5/15 etc.).
3. **Anchor selection:** choose valid bars for label creation (masking spikes/tail and requiring AEA triggers).
4. **Label generation:** for each valid anchor and each (TP, SL, lookahead), compute time‑to‑hit and label (tp\_X\_long/short, sl\_Y\_long/short).
5. **Chunking & Training:** write labels to labeliai/partN.csv; train XGBoost on chunks; checkpoint improved models; create feature‑importance summaries.
6. **Mastery roll‑up:** aggregate top‑features and hit stats per label into mastery CSVs.
7. **Black‑box opinion:** use model predict\_proba with decision gates to emit EXECUTE/IGNORE, SHAP diagnostics, and winrate summaries.
8. **Stage‑2 simulation:** apply per‑bar decisions to simulate trades with fees/leverage, create simuliacijos\_outputas\_\* and preds\_log\_\*, then generate feedback boosts.

**3) Core Parameters (tunable)**

* **Market & bars:** 1‑minute BTC/USDT, columns: open, high, low, close, volume + OF features.
* **Trading math:** initial\_balance, leverage, fee, max\_used\_balance (caps position sizing in sim/labels).
* **Label grid:** tp\_values, sl\_values, lookahead\_values (combinatorial set).
* **Compute:** n\_jobs, batch\_size, MAX\_RAM\_GB.
* **Gating:** per‑bar **AEA** flags (quality, spikes, trend/momentum, OF‑5/OF‑15, accumulation/distribution) → entry\_long / entry\_short signals.

**4) Key Components**

**4.1 AEA Signals & Valid Anchors**

* compute\_aea\_signals(df) derives quality and regime flags (spikes, ADX/RSI/Stoch, OF windows, accumulation/distribution) and terminal triggers (entry\_long, entry\_short).
* print\_aea\_stats(df, tag) prints density of each flag for quick sanity‑checks.
* count\_valid\_idx(df, lookahead) masks spike zones, tail bars, and requires AEA triggers; returns the number of usable anchors for labeling.

**4.2 Label Generation (per (TP, SL, lookahead))**

* For each valid anchor:
  + Compute **entry price** (avg of current bar high/low unless overridden) and derive **gross** price moves to reach TP/SL at target **net** P&L after **fees** and **leverage**.
  + Track earliest hit within lookahead bars; emit label and time\_to\_hit.
* Labels are direction‑aware (\*\_long/\*\_short) using AEA side locks and triggers.

**4.3 Training & Checkpointing**

* train\_on\_file(path, tp, sl, lookahead) loads a label chunk, filters unknown labels, fits an XGBClassifier with class‑encoding; logs accuracy; saves improved checkpoints to checkpoints/.
* A small "reward" mapping (+X for tp\_X, −Y for sl\_Y) is computed for optional custom metrics.
* create\_readable\_summary(model, feature\_names, …) extracts top‑N features per label for human‑readable analysis ("žmogystai").

**4.4 Mastery (roll‑up)**

* generate\_mastery(zmogystai=…, chunk\_folder=…, output\_path=…) aggregates across label chunks to compute per‑label totals: TP/SL hits, total occurrences, win‑rate, average time‑to‑hit, plus combined long/short summaries, and saves a mastery\_\*.csv.

**4.5 Black‑Box Opinion**

* black\_box\_opinion(model, X, y\_true, df\_original, model\_tp, model\_sl, threshold, class\_names, output\_path) decodes predictions, gates them by confidence **and** bar‑level gate\_long/gate\_short, computes stats per label, and dumps a CSV.
* Includes **SHAP** explainability with a dynamic threshold to filter noisy attributions and prints SHAP array diagnostics (shape per class).
* Maintains best\_blackbox\_opinion.txt with a crude coefficient for “best run” tracking.

**4.6 Stage‑2 Simulation & Feedback**

* run\_stage2(file\_path, tp, sl, initial\_balance, leverage, fee, model\_path, threshold) loads the best checkpoint, enforces feature presence/types, and generates executable signals.
* Produces: preds\_log\_tpX\_slY.csv, simuliacijos\_outputas\_tpX\_slY.csv (trade list with entry/exit bar idx, reason, PnL%), and feedback\_bank\_tpX\_slY.csv (per‑bar boost = 1.2 for wins, 0.8 for losses; neutral otherwise).
* plot\_box\_steps(…) overlays AEA boxes, entries, and exits for visual checks (optional).

**4.7 Infra Utilities**

* tqdm\_updater(q, total\_steps) renders a global progress bar as label rows are generated by workers.
* RAM footprint helpers (estimate\_gb, system\_used\_ram\_gb).

**5) Files In / Out (quick index)**

**Inputs**

* WITH\_BTC\_USDT\_1m\_part\*.csv – feature‑rich candles from Stage‑1 (ccxt + OF merge).
* (Optional) OOS\_FILE – out‑of‑sample part for preview plots.

**Models / Encoders**

* encoder.joblib – global class encoder for all TP/SL labels.
* checkpoints/model\_part{N}.joblib – best‑so‑far models per trained chunk.

**Labels & Training Artifacts**

* labeliai/tp{tp}\_sl{sl}\_look{L}\_part{N}.csv – generated labels.
* mastery\_tp{tp}\_sl{sl}\_look{L}\_part{N}.csv – mastery roll‑ups.
* black\_box\_opinion\_tp{tp}\_sl{sl}\_look{L}\_part{N}.csv – execution stats + confidence.

**Stage‑2 Simulation**

* preds\_log\_tp{tp}\_sl{sl}.csv – per‑bar predictions with confidences.
* simuliacijos\_outputas\_tp{tp}\_sl{sl}.csv – executed trades with reasons & PnL.
* feedback\_bank\_tp{tp}\_sl{sl}.csv – boosts per bar\_idx for curriculum‑style weighting.

**6) How to Run**

**6.1 Quick Start (single machine)**

1. Ensure **Stage‑1** produced WITH\_BTC\_USDT\_1m\_part\*.csv feature files.
2. Set training grid near the top:
   * tp\_values = [0.10], sl\_values = [0.10], lookahead\_values = [20] (example).
   * n\_jobs, batch\_size per your CPU/RAM.
3. Execute **AI2D.py**. It will:
   * Fit a global encoder.joblib for all classes;
   * Parallelize over (tp, sl, lookahead) with joblib (backend loky);
   * Generate labels → train → checkpoint → mastery → black\_box\_opinion.
4. Optionally call run\_stage2() (already invoked inside the training loop for OOS checks) to generate sim outputs for selected TP/SL.

**6.2 Environment Notes**

* Python 3.11+ recommended; requires: xgboost, scikit‑learn, pandas, numpy, joblib, tqdm, pympler, psutil, scipy, shap, matplotlib.
* On Windows, freeze\_support() is used for safe multiprocessing.

**7) Trading/Label Math (essentials)**

* **Net→gross conversion:** SL/TP price distances are computed to achieve **net** PnL after fees; because the strategy uses leverage and caps max\_used\_balance, we compute **position value** = used\_balance \* leverage, add **fee\_total** = position\_value \* fee \* 2 for round‑trip, and derive required price moves for TP/SL.
* **Direction logic:** entry side comes from AEA triggers and regime bias (accumulation/distribution) with spike filters and body‑size caps.

**8) Operational Tips**

* Keep encoder.joblib in sync with the **full class universe** you intend to train; regenerate if you add/remove TP/SL levels.
* If you change features, update the **entry\_features** list in Stage‑2 to avoid “missing columns” errors.
* Use SHOW\_BOX\_PREVIEW=True to visually sanity‑check AEA boxes vs. executed trades.
* Monitor best\_blackbox\_opinion.txt and checkpoints/best\_metric\_\*.txt to track improvements.
* Mastery roll‑ups are most useful when you aggregate across many chunks/parts.

**9) Minimal Example (pseudo‑flow)**

for tp in tp\_values:

for sl in sl\_values:

for L in lookahead\_values:

# Label + train + artifacts

generate\_labels\_and\_train(tp, sl, L)

# Then stage‑2 on OOS

run\_stage2("WITH\_BTC\_USDT\_1m\_part2.csv", tp=0.10, sl=0.10)

This extended section should give you a clear, reproducible spec for **AI2D.py** and how it plugs into the broader pipeline.

**Author: Lukas Svešnikovas**