```
import pandas as pd

df = pd.read_pickle('/data/workspace_files/csi_data.pkl')
df.head(-5)
```

| | | KMID | KLEN | KMID2 | KUP | KUP2 | KLOW | KLOW2 | KSFT | KSFT2 |
|---|---|---|---|---|---|---|---|---|---|---|
| datetime | instrument | | | | | | | | | |
| 2008-01-02 | SH600000 | 0.571643 | 1.800068 | 0.253804 | 3.000000 | 0.947448 | 2.712642 | 0.656584 | 0.216772 | 0.095455 |
| | SH600004 | 3.000000 | 1.715257 | 1.435924 | -0.632512 | -0.887051 | -1.055251 | -1.075074 | 2.626281 | 1.184890 |
| | SH600006 | 0.698338 | 0.598130 | 0.470045 | 0.197643 | -0.192792 | 2.230872 | 1.214465 | 1.157374 | 0.772598 |
| | SH600007 | 3.000000 | 3.000000 | 1.189876 | 0.174635 | -0.652805 | 0.505547 | -0.554218 | 3.000000 | 1.045781 |
| | SH600008 | 2.819362 | 3.000000 | 0.929474 | 3.000000 | 0.409912 | -0.508966 | -0.888653 | 1.268276 | 0.414676 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2020-09-23 | SZ300136 | 0.383612 | 0.178971 | 0.314866 | 0.900780 | 0.628875 | 1.163969 | 0.810415 | 0.349671 | 0.284643 |
| | SZ300142 | 3.000000 | 3.000000 | 0.743203 | 3.000000 | -0.092718 | 3.000000 | 0.244771 | 3.000000 | 0.713168 |
| | SZ300144 | -0.030681 | 0.102182 | -0.026238 | 0.808896 | 0.613658 | 2.121087 | 1.736632 | 0.293000 | 0.248501 |
| | SZ300347 | 1.239027 | 0.059950 | 1.084561 | 0.084450 | -0.040581 | -0.848530 | -0.887770 | 0.785844 | 0.682206 |
| | SZ300408 | -0.277211 | -0.340455 | -0.312514 | 0.436783 | 0.687468 | 0.512275 | 0.754138 | -0.257838 | -0.28827 |

870616 rows × 159 columns

```
# hirevae_train_eval.py
# Compact HiReVAE-style model for cross-sectional return prediction with on-
line regime switching.
# - Loads a Qlib-style MultiIndex df (datetime, instrument); last column is tar-
get return.
# - Trains with hierarchical VAE losses + regime separation; saves the best mod-
el on val RankIC.
# - Evaluates on test and reports RankIC and RankICIR (ICIR = mean/std).

import os
import math
import json
import random
from datetime import datetime
from typing import Tuple, Dict, List
```

```
import numpy as np
import pandas as pd
from scipy.stats import spearmanr

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

# ----------------------------
# Config (edit here)
# ----------------------------
DATA_PATH = "/data/workspace_files/csi_data.pkl"   # your pickle path
TARGET_IS_LAST_COL = True
# Time split (edit dates to your dataset); uses string compare on date level
TRAIN_END   = "2016-12-31"
VAL_END     = "2018-12-31"
# Everything after VAL_END goes to TEST

BATCH_DATES_PER_STEP = 1    # 1 date per optimizer step (keeps market coherent)
NUM_EPOCHS = 10
LR = 1e-3
WEIGHT_DECAY = 1e-5
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
SEED = 42

# Model/Regime sizes
H_MARKET = 16
H_STOCK  = 32
REGIMES  = 3          # K regimes as in the paper examples
EMA_BETA = 0.97       # history weight for regime centers (Algorithm 1, step 6)
RECON_ALPHA = 1.0     # L_rec weight
HIER_ALPHA  = 1.0     # L_hier weight
REG_ALPHA   = 1e-3    # L_reg (KL separation) weight

CHECKPOINT_PATH = "best_hirevae.pt"

# ----------------------------
# Repro
# ----------------------------
def set_seed(s=SEED):
    random.seed(s); np.random.seed(s); torch.manual_seed(s); torch.cuda.manu-
al_seed_all(s)

set_seed()

# ----------------------------
```

```
# Data prep
# ----------------------------
def load_df(path: str) -> pd.DataFrame:
    df = pd.read_pickle(path)
    if isinstance(df.index, pd.MultiIndex):
        # Ensure correct names
        if df.index.names != ["datetime", "instrument"]:
            df.index.set_names(["datetime", "instrument"], inplace=True)
    else:
        raise ValueError("Expected MultiIndex (datetime, instrument).")
    # Coerce datetime level to pd.Timestamp (date only)
    lvl0 = pd.to_datetime(df.index.get_level_values(0)).floor("D")
    df = df.copy()
    df.index = pd.MultiIndex.from_arrays([lvl0, df.index.get_level_values(1)], names=
["datetime","instrument"])
    return df.sort_index()

def split_by_date(df: pd.DataFrame, train_end: str, val_end: str) -> Tuple[pd.Data-
Frame, pd.DataFrame, pd.DataFrame]:
    dt = df.index.get_level_values(0).strftime("%Y-%m-%d")
    mask_train = dt <= train_end
    mask_val   = (dt > train_end) & (dt <= val_end)
    mask_test  = dt > val_end
    return df[mask_train], df[mask_val], df[mask_test]

def infer_cols(df: pd.DataFrame) -> Tuple[List[str], str]:
    cols = list(df.columns)
    if TARGET_IS_LAST_COL:
        target = cols[-1]
        feats = cols[:-1]
    else:
        # fallback to a conventional name
        target = "return"
        if target not in df.columns:
            raise ValueError("TARGET_IS_LAST_COL=False but no 'return' column found."
)
        feats = [c for c in cols if c != target]
    return feats, target

def build_market_features(x_cs: pd.DataFrame) -> np.ndarray:
    """
    Build a compact market feature vector from cross-section at a date.
    Here we use mean and std across stocks for each feature and then a small PCA-
like compress via learned linear layer later.
    Simpler: aggregate with mean and std over features (two summaries) to keep dimen-
sion manageable.
    """
```

```
    mu = x_cs.mean(axis=0).values
    sd = x_cs.std(axis=0).replace(0, 1e-6).values
    # concatenate; later a linear layer will reduce
    return np.concatenate([mu, sd], axis=0)

class CrossSectionDataset(Dataset):
    """
    Yields (date, X_stocks[T x F? no -> per stock], y_stocks, mar-
ket_vec) per unique date.
    We return tensors shaped (N_stocks, F) and (N_stocks,)
    """
    def __init__(self, df: pd.DataFrame, feat_cols: List[str], target_col: str):
        self.df = df
        self.feat_cols = feat_cols
        self.target_col = target_col
        self.dates = sorted(df.index.get_level_values(0).unique())
        # Precompute market features per date
        self.market_by_date: Dict[pd.Timestamp, np.ndarray] = {}
        for d in self.dates:
            cs = df.xs(d, level=0)
            self.market_by_date[d] = build_market_features(cs[self.feat_cols])

    def __len__(self):
        return len(self.dates)

    def __getitem__(self, idx):
        d = self.dates[idx]
        cs = self.df.xs(d, level=0)
        x = cs[self.feat_cols].values.astype(np.float32)
        y = cs[self.target_col].values.astype(np.float32)
        market = self.market_by_date[d].astype(np.float32)
        return d, torch.from_numpy(x), torch.from_numpy(y), torch.from_numpy(market)

def collate_dates(batch):
    # Keep dates as list, batch size = number of dates per step
    dates, Xs, Ys, Ms = zip(*batch)
    # Since we step one date at a time, just return list
    return dates, Xs, Ys, Ms

# ----------------------------
# Model
# ----------------------------
class MarketEncoder(nn.Module):
    def __init__(self, m_in, h_market):
        super().__init__()
        self.net_post = nn.Sequential(
            nn.Linear(m_in + 1, 64), nn.ReLU(),
```

```
            nn.Linear(64, 2*h_market)
        )
        self.net_prior = nn.Sequential(
            nn.Linear(m_in, 64), nn.ReLU(),
            nn.Linear(64, 2*h_market)
        )

    def forward_post(self, v, ybar):  # training (posterior uses avg future return)
        inp = torch.cat([v, ybar], dim=-1)
        out = self.net_post(inp)
        mu, logstd = out.chunk(2, dim=-1)
        return mu, logstd.clamp(-5, 3)

    def forward_prior(self, v):       # inference
        out = self.net_prior(v)
        mu, logstd = out.chunk(2, dim=-1)
        return mu, logstd.clamp(-5, 3)

class StockEncoder(nn.Module):
    def __init__(self, x_in, h_market, h_stock):
        super().__init__()
        self.net_post = nn.Sequential(
            nn.Linear(x_in + h_market + 1, 128), nn.ReLU(),
            nn.Linear(128, 2*h_stock)
        )
        self.net_prior = nn.Sequential(
            nn.Linear(x_in + h_market, 128), nn.ReLU(),
            nn.Linear(128, 2*h_stock)
        )

    def forward_post(self, x, m, y):
        inp = torch.cat([x, m, y], dim=-1)
        out = self.net_post(inp)
        mu, logstd = out.chunk(2, dim=-1)
        return mu, logstd.clamp(-5, 3)

    def forward_prior(self, x, m):
        inp = torch.cat([x, m], dim=-1)
        out = self.net_prior(inp)
        mu, logstd = out.chunk(2, dim=-1)
        return mu, logstd.clamp(-5, 3)

class RegimeSwitcher(nn.Module):
    """
    Projects market latent m -> scalar s, maintains EMA regime centers (mu_r, sig-
ma_r),
    and provides regime assignment via max log-likelihood under K Gaussians.
```

```
    """
    def __init__(self, h_market, K=3, ema_beta=0.97):
        super().__init__()
        self.proj = nn.Linear(h_market, 1)
        self.K = K
        self.register_buffer("mu_r", torch.linspace(1, -1, K).view(1, K))      # [1,K
]
        self.register_buffer("sigma_r", torch.ones(1, K))                      # [1,K
]
        self.ema_beta = ema_beta

    @staticmethod
    def _sort_desc(mu, sigma):
        idx = torch.argsort(mu, dim=-1, descending=True)
        mu_sorted = torch.gather(mu, 1, idx)
        sigma_sorted = torch.gather(sigma, 1, idx)
        return mu_sorted, sigma_sorted

    def update_centers(self, mu_batch, sigma_batch):
        mu_sorted, sigma_sorted = self._sort_desc(mu_batch, sigma_batch)
        with torch.no_grad():
            self.mu_r = self.ema_beta * self.mu_r + (1 - self.ema_beta) * mu_sorted
            self.sigma_r = self.ema_beta * self.sigma_r + (1 - self.ema_beta) * sig-
ma_sorted

    def forward_scores(self, m):
        # m: [B, Hm] (B = number of dates in step)
        s = self.proj(m)  # [B,1]
        return s

    def gaussian_params_from_m(self, m):
        # Predict batch-wise center proposals (Algorithm 1 step 4); linear head
        w_mu = nn.functional.normalize(self.proj.weight, dim=-1)  # re-
use proj shape idea; simple linear map
        # simple distinct params via two linear heads:
        mu = torch.tanh(m @ w_mu.t())     # [B,1] -> tile K
        sigma = torch.sigmoid(m @ w_mu.t()) * 0.5 + 0.1
        mu = mu.repeat(1, self.K)
        sigma = sigma.repeat(1, self.K)
        return mu, sigma

    def regime_loglik(self, s):
        # s: [B,1], centers: [1,K]
        # log N(s | mu_r, sigma_r)
        mu = self.mu_r  # [1,K]
        sig = self.sigma_r.clamp(min=1e-3)
        ll = -torch.log(sig) - 0.5*math.log(2*math.pi) - 0.5*((s -
```

```
mu)/sig)**2  # [B,K]
        return ll

    def assign(self, s):
        ll = self.regime_loglik(s)  # [B,K]
        c = torch.argmax(ll, dim=-1)  # [B]
        return c, ll

    def separation_kl(self):
        # Sum KL between each pair of regime Gaussians (Algorithm 1 text -> Lreg)
        mu = self.mu_r.squeeze(0)
        sig = self.sigma_r.squeeze(0).clamp(min=1e-3)
        total = 0.0
        for i in range(self.K):
            for j in range(i+1, self.K):
                # KL N_i || N_j (1D)
                term = torch.log(sig[j]/sig[i]) + (sig[i]**2 + (mu[i]-
mu[j])**2)/(2*sig[j]**2) - 0.5
                total += term
        return -total  # negative sum as in paper (maximize separation)

class RegimeDecoders(nn.Module):
    def __init__(self, x_in, h_market, h_stock, K):
        super().__init__()
        self.K = K
        self.decoders = nn.ModuleList([
            nn.Sequential(
                nn.Linear(x_in + h_market + h_stock, 128), nn.ReLU(),
                nn.Linear(128, 1)
            ) for _ in range(K)
        ])

    def forward(self, x, m, z, regime_idx):
        # x: [N,F], m:[N,Hm], z:[N,Hs], regime_idx scalar int
        inp = torch.cat([x, m, z], dim=-1)
        return self.decoders[regime_idx](inp).squeeze(-1)

class HiReVAE(nn.Module):
    def __init__(self, x_in: int, m_in: int, h_market=16, h_stock=32, regimes=3, ema_
beta=0.97):
        super().__init__()
        self.market_enc = MarketEncoder(m_in, h_market)
        self.stock_enc  = StockEncoder(x_in, h_market, h_stock)
        self.switcher   = RegimeSwitcher(h_market, K=regimes, ema_beta=ema_beta)
        self.decoders   = RegimeDecoders(x_in, h_market, h_stock, regimes)
        # reduce market feature dimension
        self.m_reduce = nn.Sequential(nn.Linear(m_in, 64), nn.ReLU(), nn.Linear(64, m
```

```
_in))

    @staticmethod
    def _rsample(mu, logstd):
        eps = torch.randn_like(mu)
        return mu + eps * torch.exp(logstd)

    def forward_train_one_date(self, X, y, v):
        N, D = X.shape       # ← was: N, F = X.shape  (rename avoids shadowing)
        v = v.unsqueeze(0)
        v = self.m_reduce(v)

        ybar = y.mean().view(1,1)
        mu_m_post, logstd_m_post = self.market_enc.forward_post(v, ybar)
        m_post = self._rsample(mu_m_post, logstd_m_post)

        mu_cand, sigma_cand = self.switcher.gaussian_params_from_m(m_post)
        self.switcher.update_centers(mu_cand.detach(), sigma_cand.detach())

        s = self.switcher.forward_scores(m_post)
        c, ll = self.switcher.assign(s)
        regime_idx = int(c.item())

        m_tiled = m_post.expand(N, -1)
        mu_z_post, logstd_z_post = self.stock_enc.forward_post(X, m_tiled, y.un-
squeeze(-1))
        z_post = self._rsample(mu_z_post, logstd_z_post)

        mu_m_prior, logstd_m_prior = self.market_enc.forward_prior(v)
        mu_z_prior, logstd_z_prior = self.stock_enc.forward_prior(X, m_tiled)

        y_hat = self.decoders(X, m_tiled, z_post, regime_idx)

        rec = F.mse_loss(y_hat, y)  # now F refers to torch.nn.functional again

        def kl_norm(mu_q, logstd_q, mu_p, logstd_p):
            var_q = torch.exp(2*logstd_q)
            var_p = torch.exp(2*logstd_p)
            kld = ((var_q + (mu_q - mu_p)**2) / var_p - 1.0) + 2*(logstd_p - logst-
d_q)
            return 0.5 * kld.sum(dim=-1)

        kl_m = kl_norm(mu_m_post, logstd_m_post, mu_m_prior, logstd_m_prior).mean()
        kl_z = kl_norm(mu_z_post, logstd_z_post, mu_z_prior, logstd_z_prior).mean()
        hier = kl_m + kl_z
        reg = self.switcher.separation_kl()
```

```
        loss = RECON_ALPHA*rec + HIER_ALPHA*hier + REG_ALPHA*reg
        return loss, rec.detach(), hier.detach(), reg.detach(), regime_idx, y_hat.de-
tach()

    @torch.no_grad()
    def predict_one_date(self, X, v):
        """
        Inference: use priors only and choose decoder by regime from prior market la-
tent.
        """
        N, F = X.shape
        v = v.unsqueeze(0)
        v = self.m_reduce(v)
        mu_m0, logstd_m0 = self.market_enc.forward_prior(v)
        m0 = mu_m0  # use mean of prior (deterministic)
        s = self.switcher.forward_scores(m0)
        c, _ = self.switcher.assign(s)
        regime_idx = int(c.item())
        m_tiled = m0.expand(N, -1)
        mu_z0, logstd_z0 = self.stock_enc.forward_prior(X, m_tiled)
        z0 = mu_z0
        y_hat = self.decoders(X, m_tiled, z0, regime_idx)
        return y_hat.cpu().numpy(), regime_idx

# ----------------------------
# Training / Evaluation utils
# ----------------------------
def spearman_rank_ic(pred: np.ndarray, truth: np.ndarray) -> float:
    if np.all(np.isfinite(pred)) and np.all(np.isfinite(truth)) and pred.size > 1:
        rho, _ = spearmanr(pred, truth)
        return float(rho)
    return 0.0

def evaluate_rankic(model: HiReVAE, ds: CrossSectionDataset, dates_subset=None) -
> Tuple[float,float]:
    model.eval()
    dates = dates_subset if dates_subset is not None else ds.dates
    ics = []
    for d in dates:
        _, X, y, v = ds[ds.dates.index(d)]
        X = X.to(DEVICE); v = v.to(DEVICE)
        y_hat, _ = model.predict_one_date(X, v)
        ic = spearman_rank_ic(y_hat, y.numpy())
        if np.isfinite(ic):
            ics.append(ic)
    if len(ics)==0:
        return 0.0, 0.0
```

```
    ics = np.array(ics)
    rankic = float(np.nanmean(ics))
    icir   = float(rankic/ (np.nanstd(ics, ddof=1)+1e-12))
    return rankic, icir

def train(model, dl, val_ds):
    opt = torch.optim.AdamW(model.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)
    best_val = -1e9
    for epoch in range(1, NUM_EPOCHS+1):
        model.train()
        losses, recs, hiers, regs = [], [], [], []
        for dates, Xs, Ys, Ms in dl:
            # Train per date to keep market latent coherent
            opt.zero_grad()
            assert len(dates)==1, "Use BATCH_DATES_PER_STEP=1 for coherence."
            X, y, v = Xs[0].to(DEVICE), Ys[0].to(DEVICE), Ms[0].to(DEVICE)
            loss, rec, hier, reg, regime_idx, _ = model.forward_train_one_-
date(X, y, v)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
            losses.append(loss.item()); recs.append(rec.item()); hiers.append(hi-
er.item()); regs.append(reg.item())

        val_rankic, val_icir = evaluate_rankic(model, val_ds)
        print(f"Epoch {epoch}/{NUM_EPOCHS} "
              f"| loss {np.mean(losses):.4f} | rec {np.mean(recs):.4f} |
 hier {np.mean(hiers):.4f} | reg {np.mean(regs):.4f} "
              f"| VAL RankIC {val_rankic:.4f} | VAL ICIR {val_icir:.4f}")

        # Save best on RankIC
        if val_rankic > best_val:
            best_val = val_rankic
            torch.save({"model_state": model.state_dict(),
                        "meta": {"epoch": epoch, "val_rankic": val_rankic, "val_icir"
: val_icir}},
                       CHECKPOINT_PATH)
            print(f">>> Saved new best model to {CHECKPOINT_PATH} (VAL RankIC {val_ra
nkic:.4f})")
    return best_val

# ----------------------------
# Main
# ----------------------------
def main():
    print("Loading data...")
    df = load_df(DATA_PATH)
```

```
    feat_cols, target_col = infer_cols(df)

    # Drop any NaNs (simple robustification)
    df = df.replace([np.inf, -np.inf], np.nan).dropna(subset=feat_cols + [target_-
col])

    df_tr, df_val, df_te = split_by_date(df, TRAIN_END, VAL_END)
    print(f"Train dates: {df_tr.index.get_level_values(0).unique().min()} -
> {df_tr.index.get_level_values(0).unique().max()} "
          f"({len(df_tr.index.get_level_values(0).unique())} days)")
    print(f"Val   dates: {df_val.index.get_level_values(0).unique().min()} -> {df_-
val.index.get_level_values(0).unique().max()} "
          f"({len(df_val.index.get_level_values(0).unique())} days)")
    print(f"Test  dates: {df_te.index.get_level_values(0).unique().min()} -
> {df_te.index.get_level_values(0).unique().max()} "
          f"({len(df_te.index.get_level_values(0).unique())} days)")

    train_ds = CrossSectionDataset(df_tr, feat_cols, target_col)
    val_ds   = CrossSectionDataset(df_val, feat_cols, target_col)
    test_ds  = CrossSectionDataset(df_te, feat_cols, target_col)

    m_in = train_ds.market_by_date[train_ds.dates[0]].shape[0]
    FEAT_DIM = len(feat_cols)            # ← was F = len(feat_cols)
    model = HiReVAE(x_in=FEAT_DIM, m_in=m_in, h_market=H_MARKET, h_stock=H_STOCK,
                    regimes=REGIMES, ema_beta=EMA_BETA).to(DEVICE)

    train_loader = DataLoader(train_ds, batch_size=BATCH_DATES_PER_STEP, shuffle=Fals
e, collate_fn=collate_dates)

    print("Training...")
    best = train(model, train_loader, val_ds)

    # Load best and evaluate
    if os.path.exists(CHECKPOINT_PATH):
        ckpt = torch.load(CHECKPOINT_PATH, map_location=DEVICE)
        model.load_state_dict(ckpt["model_state"])
        print(f"Loaded best checkpoint (epoch={ckpt['meta']['epoch']}, val_rankic=
{ckpt['meta']['val_rankic']:.4f})")

    print("Evaluating (validation + test) with priors only...")
    val_rankic, val_icir = evaluate_rankic(model, val_ds)
    te_rankic, te_icir = evaluate_rankic(model, test_ds)
    print(json.dumps({
        "val_rankic": round(val_rankic, 6),
        "val_icir": round(val_icir, 6),
        "test_rankic": round(te_rankic, 6),
        "test_icir": round(te_icir, 6),
```

```
    }, indent=2))

if __name__ == "__main__":
    main()
```

```
Loading data...
Train dates: 2008-01-02 00:00:00 -> 2016-12-30 00:00:00 (2140 days)
Val   dates: 2017-01-03 00:00:00 -> 2018-12-28 00:00:00 (487 days)
Test  dates: 2019-01-02 00:00:00 -> 2020-09-23 00:00:00 (419 days)
Training...
Epoch 1/10 | loss 1.0724 | rec 1.0043 | hier 0.0682 | reg -0.0442 | VAL RankIC 0.0522
>>> Saved new best model to best_hirevae.pt (VAL RankIC 0.0522)
Epoch 2/10 | loss 0.9936 | rec 0.9932 | hier 0.0004 | reg 0.0000 | VAL RankIC 0.0608 |
>>> Saved new best model to best_hirevae.pt (VAL RankIC 0.0608)
Epoch 3/10 | loss 0.9919 | rec 0.9915 | hier 0.0004 | reg 0.0000 | VAL RankIC 0.0632 |
>>> Saved new best model to best_hirevae.pt (VAL RankIC 0.0632)
Epoch 4/10 | loss 0.9908 | rec 0.9904 | hier 0.0004 | reg 0.0000 | VAL RankIC 0.0674 |
>>> Saved new best model to best_hirevae.pt (VAL RankIC 0.0674)
Epoch 5/10 | loss 0.9902 | rec 0.9899 | hier 0.0003 | reg 0.0000 | VAL RankIC 0.0699 |
>>> Saved new best model to best_hirevae.pt (VAL RankIC 0.0699)
Epoch 6/10 | loss 0.9897 | rec 0.9892 | hier 0.0004 | reg 0.0000 | VAL RankIC 0.0662 |
Epoch 7/10 | loss 0.9892 | rec 0.9889 | hier 0.0003 | reg 0.0000 | VAL RankIC 0.0670 |
Epoch 8/10 | loss 0.9889 | rec 0.9884 | hier 0.0005 | reg 0.0000 | VAL RankIC 0.0656 |
Epoch 9/10 | loss 0.9884 | rec 0.9881 | hier 0.0003 | reg 0.0000 | VAL RankIC 0.0665 |
Epoch 10/10 | loss 0.9881 | rec 0.9878 | hier 0.0004 | reg 0.0000 | VAL RankIC 0.0670
```

```
# === PATHS (yours) ===
DATA_PATH = "/data/workspace_files/csi_data.pkl"          # MultiIndex (datetime, in-
strument); LAST col = next-day return
CHECKPOINT_PATH = "/data/notebook_files/best_hirevae.pt"  # saved best model

import os, math, json, random, warnings
from typing import List, Tuple
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F

warnings.filterwarnings("ignore")

# ----------------- Config -----------------
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
SEED = 42
random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED); torch.cuda.manu-
al_seed_all(SEED)
```

```
# Time split (adjust to your training split if different)
TRAIN_END = "2017-12-31"
VAL_END   = "2018-12-29"
TARGET_IS_LAST_COL = True

# Matplotlib (HQ but neutral styling)
plt.rcParams.update({
    "figure.dpi": 160, "savefig.dpi": 200, "font.size": 12,
    "axes.titleweight": "semibold", "axes.grid": True, "grid.linestyle": "--
", "grid.linewidth": 0.6,
    "lines.linewidth": 2.0, "legend.frameon": False, "figure.autolayout": True,
})

# ---------------- Data helpers ----------------
def load_df(path: str) -> pd.DataFrame:
    df = pd.read_pickle(path)
    if not isinstance(df.index, pd.MultiIndex):
        raise ValueError("Expected MultiIndex (datetime, instrument).")
    dt  = pd.to_datetime(df.index.get_level_values(0)).floor("D")
    ins = df.index.get_level_values(1).astype(str)
    df = df.copy()
    df.index = pd.MultiIndex.from_arrays([dt, ins], names=["datetime","instrument"])
    return df.sort_index()

def infer_cols(df: pd.DataFrame) -> Tuple[List[str], str]:
    cols = list(df.columns)
    if TARGET_IS_LAST_COL:
        return cols[:-1], cols[-1]
    target = "return"; assert target in df.columns, "No 'return' column found."
    return [c for c in cols if c != target], target

def split_by_date(df: pd.DataFrame, train_end: str, val_end: str):
    dts = df.index.get_level_values(0)
    tr = df[dts <= pd.to_datetime(train_end)]
    va = df[(dts > pd.to_datetime(train_end)) & (dts <= pd.to_datetime(val_end))]
    te = df[dts > pd.to_datetime(val_end)]
    return tr, va, te

def build_market_features(x_cs: pd.DataFrame) -> np.ndarray:
    mu = x_cs.mean(axis=0).values
    sd = x_cs.std(axis=0).replace(0, 1e-6).values
    return np.concatenate([mu, sd], axis=0).astype(np.float32)

def ensure_multiindex(dfp: pd.DataFrame) -> pd.DataFrame:
    if isinstance(dfp.index, pd.MultiIndex) and dfp.index.nlevels == 2:
        dt = pd.to_datetime(dfp.index.get_level_values(0)).floor("D")
```

```
        ins = dfp.index.get_level_values(1).astype(str)
        out = dfp.copy()
        out.index = pd.MultiIndex.from_arrays([dt, ins], names=["datetime","instru-
ment"])
        return out.sort_index()
    if len(dfp.index) > 0 and isinstance(dfp.index[0], tuple) and len(dfp.index[0]) =
= 2:
        dates = pd.to_datetime([t[0] for t in dfp.index]).floor("D")
        insts = [str(t[1]) for t in dfp.index]
        out = dfp.copy()
        out.index = pd.MultiIndex.from_arrays([dates, insts], names=["datetime","in-
strument"])
        return out.sort_index()
    raise TypeError("pred_df must have a MultiIndex (datetime, instrument).")

# ----------------- Model (same shapes as training) -----------------
class MarketEncoder(nn.Module):
    def __init__(self, m_in, h_market):
        super().__init__()
        self.net_prior = nn.Sequential(nn.Linear(m_in,64), nn.ReLU(), nn.Linear(64, 2
*h_market))
    def forward_prior(self, v):
        out = self.net_prior(v); mu, logstd = out.chunk(2,-1); return mu, logstd.-
clamp(-5,3)

class StockEncoder(nn.Module):
    def __init__(self, x_in, h_market, h_stock):
        super().__init__()
        self.net_prior = nn.Sequential(nn.Linear(x_in+h_market,128), nn.ReLU(), nn.-
Linear(128, 2*h_stock))
    def forward_prior(self, x, m):
        out = self.net_prior(torch.cat([x,m],-1)); mu, logstd = out.chunk(2,-1); re-
turn mu, logstd.clamp(-5,3)

class RegimeSwitcher(nn.Module):
    def __init__(self, h_market, K=3):
        super().__init__()
        self.proj = nn.Linear(h_market,1)
        self.register_buffer("mu_r", torch.linspace(1,-1,K).view(1,K))
        self.register_buffer("sigma_r", torch.ones(1,K))
    def forward_scores(self,m): return self.proj(m)
    def regime_loglik(self,s):
        mu=self.mu_r; sig=self.sigma_r.clamp(min=1e-3)
        return -torch.log(sig)-0.5*math.log(2*math*pi)-0.5*((s-mu)/sig)**2

# (fix typo in previous function: use math.pi)
from math import pi as _mpi
```

```
def _ll(m, mu, sig): return -np.log(sig) - 0.5*np.log(2*_mpi) - 0.5*((m-mu)/sig)**2

class RegimeSwitcher(nn.Module):
    def __init__(self, h_market, K=3):
        super().__init__()
        self.proj = nn.Linear(h_market,1)
        self.register_buffer("mu_r", torch.linspace(1,-1,K).view(1,K))
        self.register_buffer("sigma_r", torch.ones(1,K))
    def forward_scores(self,m): return self.proj(m)
    def regime_loglik(self,s):
        mu=self.mu_r; sig=self.sigma_r.clamp(min=1e-3)
        return -torch.log(sig)-0.5*math.log(2*math.pi)-0.5*((s-mu)/sig)**2
    def assign(self,s): ll=self.regime_loglik(s); c=torch.argmax(ll,-1); return c, ll

class RegimeDecoders(nn.Module):
    def __init__(self, x_in, h_market, h_stock, K):
        super().__init__()
        self.decoders = nn.ModuleList([nn.Sequential(nn.Linear(x_in+h_market+h_stock,
128), nn.ReLU(), nn.Linear(128,1)) for _ in range(K)])
    def forward(self,x,m,z,idx): return self.decoders[idx](torch.-
cat([x,m,z],-1)).squeeze(-1)

class HiReVAE(nn.Module):
    def __init__(self, x_in, m_in, h_market=16, h_stock=32, regimes=3):
        super().__init__()
        self.market_enc = MarketEncoder(m_in, h_market)
        self.stock_enc  = StockEncoder(x_in, h_market, h_stock)
        self.switcher   = RegimeSwitcher(h_market, regimes)
        self.decoders   = RegimeDecoders(x_in, h_market, h_stock, regimes)
        self.m_reduce   = nn.Sequential(nn.Linear(m_in,64), nn.ReLU(), nn.Linear(64,m
_in))
    @torch.no_grad()
    def predict(self, X, v):
        N,_ = X.shape
        v = v.unsqueeze(0); v = self.m_reduce(v)
        mu_m, logstd_m = self.market_enc.forward_prior(v); m = mu_m
        s = self.switcher.forward_scores(m); c,_ = self.switcher.assign(s); idx=int(c
.item())
        m_tile = m.expand(N,-1)
        mu_z, logstd_z = self.stock_enc.forward_prior(X, m_tile); z = mu_z
        y_hat = self.decoders(X, m_tile, z, idx)
        uncert = torch.exp(2*logstd_z).mean(dim=1)
        return y_hat.cpu().numpy(), uncert.cpu().numpy(), idx

# ---------------- Load data & model ----------------
assert os.path.exists(DATA_PATH), f"DATA_PATH not found: {DATA_PATH}"
assert os.path.exists(CHECKPOINT_PATH), f"CHECKPOINT_PATH not found: {CHECKPOINT_PATH
```

```
}"

df = load_df(DATA_PATH)
feat_cols, target_col = infer_cols(df)
df = df.replace([np.inf,-np.inf], np.nan).dropna(subset=feat_cols+[target_col])
df_tr, df_val, df_te = split_by_date(df, TRAIN_END, VAL_END)

def market_vec_for_date(d, subset):
    cs = subset.xs(d, level=0)
    return build_market_features(cs[feat_cols])

m_in = market_vec_for_date(df_tr.index.get_level_values(0).unique()
[0], df_tr).shape[0]
x_in = len(feat_cols)

# ---------- Load model ----------
model = HiReVAE(x_in=x_in, m_in=m_in).to(DEVICE)
ckpt = torch.load(CHECKPOINT_PATH, map_location=DEVICE)
load_res = model.load_state_dict(ckpt["model_state"], strict=False)  # relaxed load
print("Checkpoint loaded with relaxed matching.")
print("  Missing keys   :", load_res.missing_keys)
print("  Unexpected keys:", load_res.unexpected_keys)

# ---------------- Predict on TEST ----------------
test_dates = sorted(df_te.index.get_level_values(0).unique())
preds, uncs, idxs = [], [], []
for d in test_dates:
    cs = df_te.xs(d, level=0)
    X = torch.from_numpy(cs[feat_cols].values.astype(np.float32)).to(DEVICE)
    v = torch.from_numpy(market_vec_for_date(d, df_te)).to(DEVICE)
    y_hat, u_hat, _ = model.predict(X, v)
    preds.append(y_hat); uncs.append(u_hat)
    idxs.append(pd.MultiIndex.from_product([[d], cs.index], names=["datetime","in-
strument"]))

pred = pd.Series(np.concatenate(preds), index=pd.Index(np.concatenate(idxs)))
unc  = pd.Series(np.concatenate(uncs),  index=pred.index)
y    = df_te[target_col].reindex(pred.index)

pred_df = pd.DataFrame({"pred": pred, "uncert": unc, "y_true": y})
pred_df = ensure_multiindex(pred_df)

# ---------------- Strategies: daily returns ----------------
def long_only_top_k(dfp, k=50):
    vals={}
    for d in sorted(dfp.index.get_level_values(0).unique()):
        cs = dfp.xs(d, level=0)
```

```
        if cs.shape[0] < k: continue
        vals[d] = cs.nlargest(k,"pred")["y_true"].mean()
    return pd.Series(vals).sort_index()

def long_only_top_k_penalized(dfp, k=50, lam=0.5):
    vals={}
    for d in sorted(dfp.index.get_level_values(0).unique()):
        cs = dfp.xs(d, level=0).copy()
        if cs.shape[0] < k: continue
        z = (cs["uncert"] - cs["uncert"].mean())/(cs["uncert"].std(ddof=1)+1e-12)
        score = cs["pred"] - lam*z
        vals[d] = cs.assign(score=score).nlargest(k,"score")["y_true"].mean()
    return pd.Series(vals).sort_index()

def long_short_decile(dfp, q=0.10):
    vals={}
    for d in sorted(dfp.index.get_level_values(0).unique()):
        cs = dfp.xs(d, level=0)
        n = int(max(1, round(cs.shape[0]*q)))
        if cs.shape[0] < 20 or n < 1: continue
        vals[d] = cs.nlargest(n,"pred")["y_true"].mean() - cs.nsmallest(n,"pred")
["y_true"].mean()
    return pd.Series(vals).sort_index()

def long_short_decile_penalized(dfp, q=0.10, lam=0.5):
    vals={}
    for d in sorted(dfp.index.get_level_values(0).unique()):
        cs = dfp.xs(d, level=0).copy()
        n = int(max(1, round(cs.shape[0]*q)))
        if cs.shape[0] < 20 or n < 1: continue
        z = (cs["uncert"] - cs["uncert"].mean())/(cs["uncert"].std(ddof=1)+1e-12)
        score = cs["pred"] - lam*z
        vals[d] = cs.assign(score=score).nlargest(n,"score")["y_true"].mean() -
 cs.assign(score=score).nsmallest(n,"score")["y_true"].mean()
    return pd.Series(vals).sort_index()

top50         = long_only_top_k(pred_df, 50)
top50_pen     = long_only_top_k_penalized(pred_df, 50, 0.5)
ls_decile     = long_short_decile(pred_df, 0.10)
ls_decile_pen = long_short_decile_penalized(pred_df, 0.10, 0.5)

rets_df = pd.DataFrame({
    "top50": top50,
    "top50_pen": top50_pen,
    "ls_decile": ls_decile,
    "ls_decile_pen": ls_decile_pen,
}).sort_index()
```

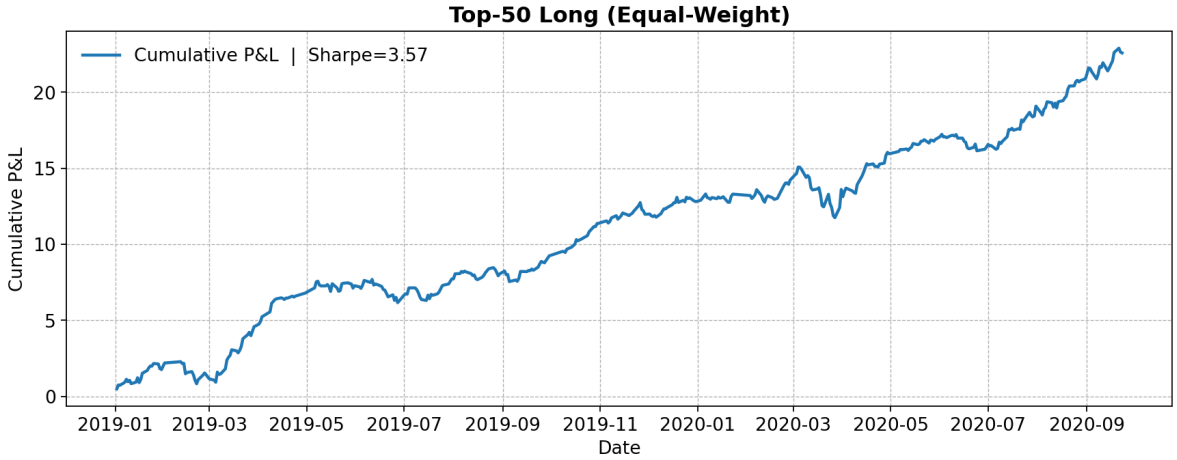
```
# ----------------- Helpers: Sharpe & Drawdown -----------------
def sharpe_annualized(series: pd.Series, periods_per_year: int = 252) -> float:
    m = series.mean()
    s = series.std(ddof=1)
    return float("nan") if s == 0 or np.isnan(s) else float((m/s) * np.sqrt(period-
s_per_year))

def equity_from_pnl(series: pd.Series) -> pd.Series:
    """Convert daily returns to 'equity' using cumulative PnL (start at 1)."""
    pnl = series.fillna(0).cumsum()
    return 1.0 + pnl

def drawdown_pct(series: pd.Series) -> pd.Series:
    """Drawdown (%) from running peak of equity curve."""
    eq = equity_from_pnl(series)
    roll_max = eq.cummax()
    dd = (eq / roll_max) - 1.0
    return 100.0 * dd  # percent

# ----------------- P&L plots (annotate Sharpe) -----------------
def plot_cum_pnl(series: pd.Series, title: str, filename: str):
    pnl = series.fillna(0).cumsum()
    sh = sharpe_annualized(series)
    plt.figure(figsize=(11, 4.5))
    plt.plot(pnl.index, pnl.values, label=f"Cumulative P&L  |  Sharpe={sh:.2f}")
    plt.title(title)
    plt.xlabel("Date"); plt.ylabel("Cumulative P&L")
    plt.legend(loc="best")
    plt.tight_layout()
    out = f"/data/workspace_files/{filename}"
    plt.savefig(out)
    plt.show()

# ----------------- Drawdown plots -----------------
def plot_drawdown(series: pd.Series, title: str, filename: str):
    dd = drawdown_pct(series)
    plt.figure(figsize=(11, 3.8))
    plt.fill_between(dd.index, dd.values, 0, step=None, alpha=0.35)
    plt.title(title + " — Drawdown (%)")
    plt.xlabel("Date"); plt.ylabel("Drawdown (%)")
    plt.tight_layout()
    out = f"/data/workspace_files/{filename}"
    plt.savefig(out)
    plt.show()

# ---- PnL + Drawdown for each strategy ----
```

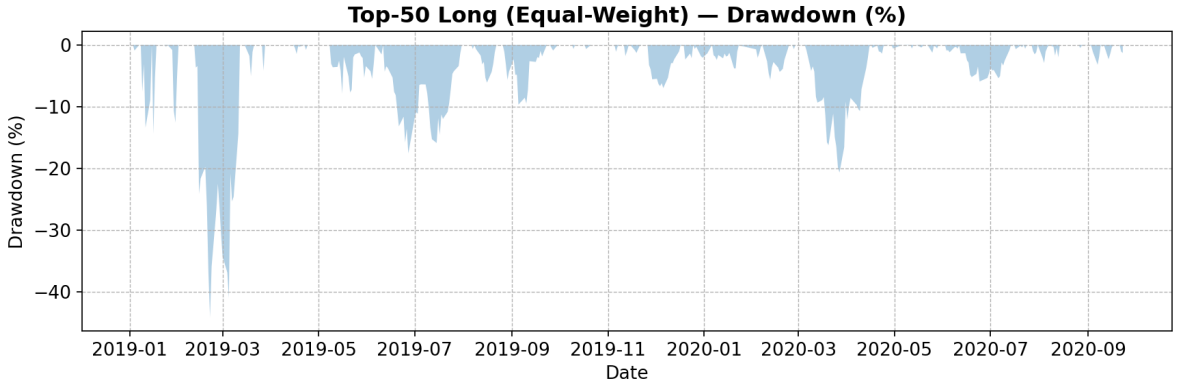
```
plot_cum_pnl(rets_df["top50"],         "Top-50 Long (Equal-
Weight)",                   "pnl_top50.png")
plot_drawdown(rets_df["top50"],        "Top-50 Long (Equal-
Weight)",                   "dd_top50.png")

plot_cum_pnl(rets_df["top50_pen"],     "Top-50 Long (Uncertainty-Penal-
ized)",           "pnl_top50_penalized.png")
plot_drawdown(rets_df["top50_pen"],    "Top-50 Long (Uncertainty-Penal-
ized)",           "dd_top50_penalized.png")

plot_cum_pnl(rets_df["ls_decile"],     "Long-
Short Decile 10%",                        "pnl_ls_decile.png")
plot_drawdown(rets_df["ls_decile"],    "Long-
Short Decile 10%",                        "dd_ls_decile.png")

plot_cum_pnl(rets_df["ls_decile_pen"], "Long-Short Decile 10% (Uncertainty-Penal-
ized)", "pnl_ls_decile_penalized.png")
plot_drawdown(rets_df["ls_decile_pen"],"Long-Short Decile 10% (Uncertainty-Penal-
ized)", "dd_ls_decile_penalized.png")

# ----------------- RankIC & Rolling IC/ICIR over time -----------------
def rank_ic_by_date(dfp: pd.DataFrame) -> pd.Series:
    vals = {}
    for d in sorted(dfp.index.get_level_values(0).unique()):
        cs = dfp.xs(d, level=0)
        if cs.shape[0] < 5:
            continue
        vals[d] = cs["pred"].rank().corr(cs["y_true"].rank(), method="spearman")
    return pd.Series(vals).sort_index()

rankic = rank_ic_by_date(pred_df)

def plot_ic_with_rolling(ic: pd.Series, win: int = 60, filename: str = "ic_time-
series.png"):
    rmean = ic.rolling(win, min_periods=max(3, win//3)).mean()
    plt.figure(figsize=(11, 3.6))
    plt.plot(ic.index, ic.values, alpha=0.6, label="Daily RankIC")
    plt.plot(rmean.index, rmean.values, linewidth=2.5, label=f"{win}d Rolling Mean")
    plt.axhline(0.0, color="k", linewidth=1.0, linestyle=":")
    plt.title("Daily Cross-Sectional RankIC Over Time")
    plt.xlabel("Date"); plt.ylabel("Spearman RankIC")
    plt.legend()
    plt.tight_layout()
    out = f"/data/workspace_files/{filename}"
    plt.savefig(out)
    plt.show()
```

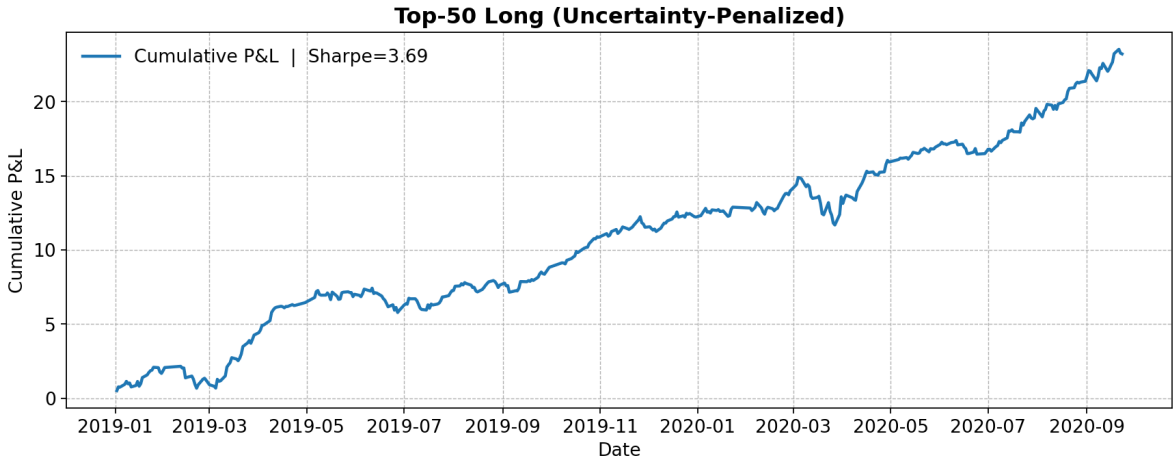
```
def plot_ic_and_icir(ic: pd.Series, win: int = 60, filename: str = "ic_icir_time-
series.png"):
    rmean = ic.rolling(win, min_periods=max(3, win//3)).mean()
    rstd  = ic.rolling(win, min_periods=max(3, win//3)).std(ddof=1)
    icir  = rmean / (rstd + 1e-12)

    fig, ax1 = plt.subplots(figsize=(11, 4.0))
    ax1.bar(ic.index, ic.values, width=3, alpha=0.25, label="Daily RankIC")
    ax1.axhline(0.0, color="k", linewidth=1.0, linestyle=":")
    ax1.set_ylabel("Daily RankIC"); ax1.set_xlabel("Date")
    ax1.set_title("Daily RankIC and Rolling ICIR Over Time")

    ax2 = ax1.twinx()
    ax2.plot(icir.index, icir.values, linewidth=2.5, label=f"Rolling ICIR ({win}d)")
    ax2.set_ylabel("Rolling ICIR")

    l1, _ = ax1.get_legend_handles_labels()
    l2, _ = ax2.get_legend_handles_labels()
    ax2.legend(l1 + l2, ["Daily RankIC", f"Rolling ICIR ({win}d)"], loc="upper left")

    fig.tight_layout()
    out = f"/data/workspace_files/{filename}"
    plt.savefig(out)
    plt.show()

plot_ic_with_rolling(rankic, win=60, filename="ic_timeseries.png")
plot_ic_and_icir(rankic, win=60, filename="ic_icir_timeseries.png")

# ----------------- Decile means with 95% confidence intervals -----------------
def decile_ci_plot(dfp: pd.DataFrame, bins: int = 10, title: str = "Avg next-day re-
turn by decile (95% CI)", filename: str = "deciles_ci.png"):
    rows = []
    dates = sorted(dfp.index.get_level_values(0).unique())
    for d in dates:
        cs = dfp.xs(d, level=0).copy()
        if cs.shape[0] < bins:
            continue
        cs["dec"] = pd.qcut(cs["pred"].rank(method="first"), bins, labels=False) + 1
        rows.append(cs.groupby("dec")["y_true"].mean())
    if not rows:
        print("Not enough cross-sections to compute decile CIs.")
        return
    frame = pd.concat(rows, axis=1).T
    mu = frame.mean(axis=0)
    sd = frame.std(axis=0, ddof=1)
    n  = frame.notna().sum(axis=0).clip(lower=1)
    se = sd / np.sqrt(n)
```

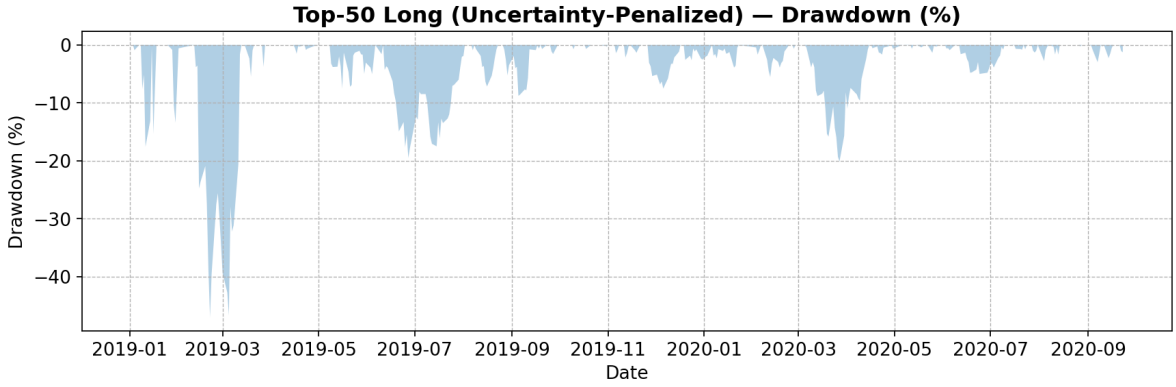
```
    ci95 = 1.96 * se

    x = mu.index.astype(int)
    plt.figure(figsize=(8.5, 4.5))
    plt.errorbar(x, mu.values, yerr=ci95.values, fmt='o', capsize=4)
    plt.title(title)
    plt.xlabel("Predicted-return decile (1=lowest, 10=highest)")
    plt.ylabel("Avg next-day return")
    plt.tight_layout()
    out = f"/data/workspace_files/{filename}"
    plt.savefig(out)
    plt.show()

decile_ci_plot(pred_df, bins=10, title="Avg next-day re-
turn by decile with 95% CI", filename="deciles_ci.png")

# ----------------- Save daily returns we used for P&L -----------------
rets_csv = "/data/workspace_files/hirevae_strategy_daily_returns_pnl.csv"
rets_df.to_csv(rets_csv)
print(json.dumps({
    "returns_csv": rets_csv,
    "plots": [
        "/data/workspace_files/pnl_top50.png",
        "/data/workspace_files/dd_top50.png",
        "/data/workspace_files/pnl_top50_penalized.png",
        "/data/workspace_files/dd_top50_penalized.png",
        "/data/workspace_files/pnl_ls_decile.png",
        "/data/workspace_files/dd_ls_decile.png",
        "/data/workspace_files/pnl_ls_decile_penalized.png",
        "/data/workspace_files/dd_ls_decile_penalized.png",
        "/data/workspace_files/ic_timeseries.png",
        "/data/workspace_files/ic_icir_timeseries.png",
        "/data/workspace_files/deciles_ci.png",
    ]
}, indent=2))
```

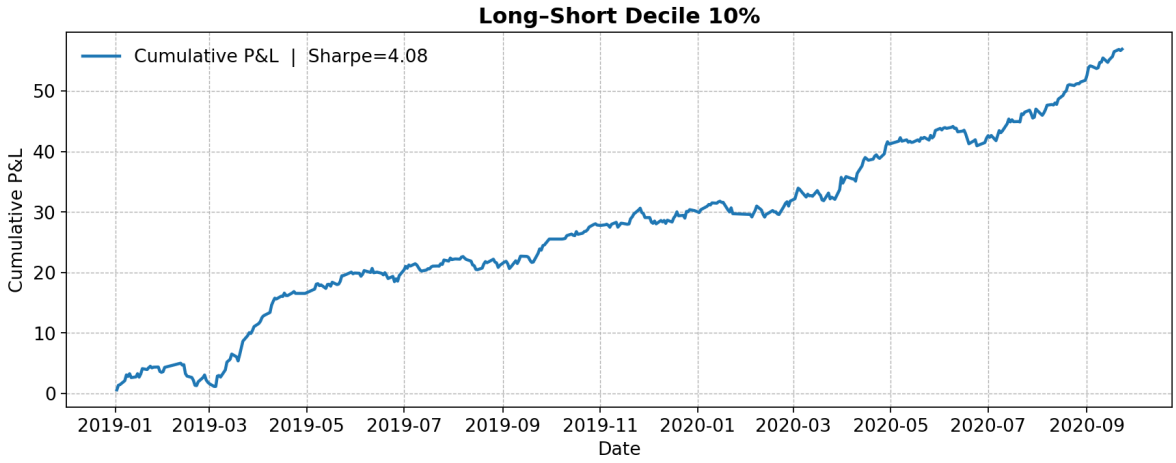
```
Checkpoint loaded with relaxed matching.
  Missing keys   : []
  Unexpected keys: ['market_enc.net_post.0.weight', 'market_enc.net_post.0.bias', 'mar
{
  "returns_csv": "/data/workspace_files/hirevae_strategy_daily_returns_pnl.csv",
  "plots": [
    "/data/workspace_files/pnl_top50.png",
    "/data/workspace_files/dd_top50.png",
    "/data/workspace_files/pnl_top50_penalized.png",
    "/data/workspace_files/dd_top50_penalized.png",
    "/data/workspace_files/pnl_ls_decile.png",
    "/data/workspace_files/dd_ls_decile.png",
    "/data/workspace_files/pnl_ls_decile_penalized.png",
    "/data/workspace_files/dd_ls_decile_penalized.png",
    "/data/workspace_files/ic_timeseries.png",
    "/data/workspace_files/ic_icir_timeseries.png",
    "/data/workspace_files/deciles_ci.png"
  ]
}
```

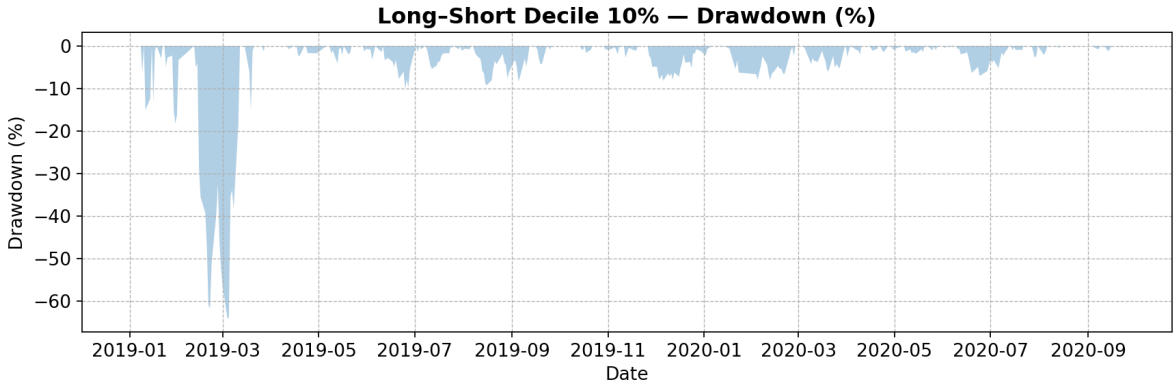


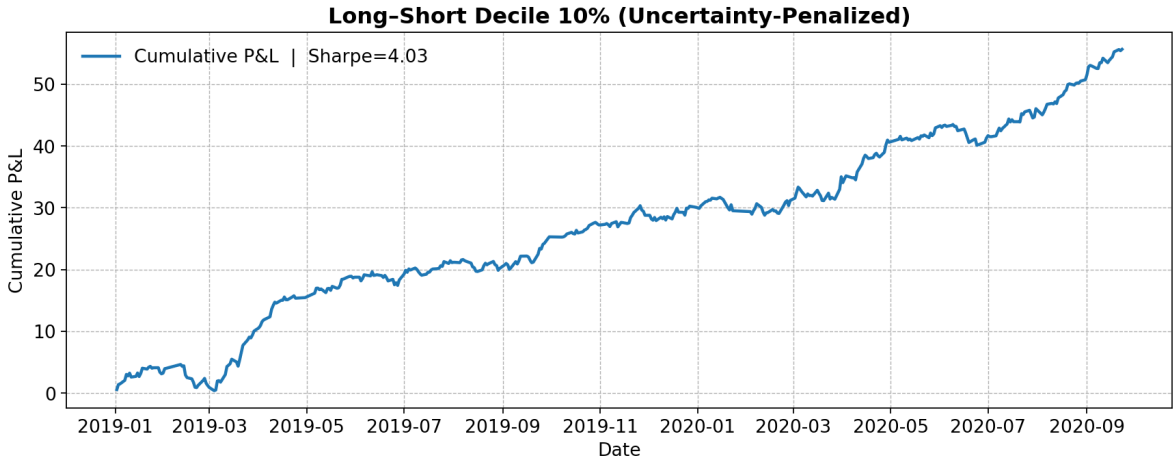



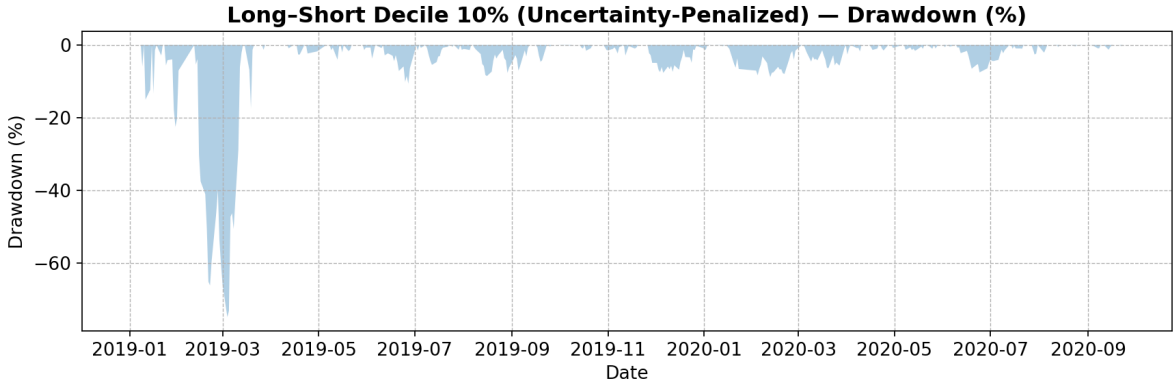

Top-50 Long (Uncertainty-Penalized)
Cumulative P&L | Sharpe=3.69
Cumulative P&L
Date

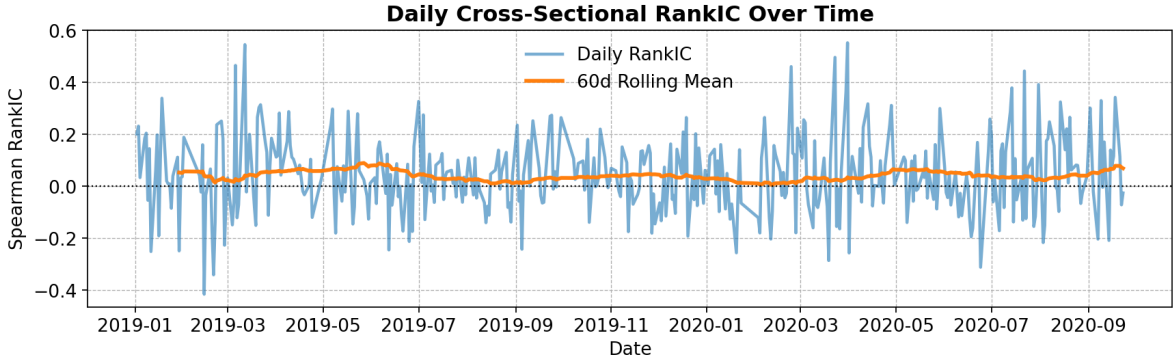

Top-50 Long (Uncertainty-Penalized) — Drawdown (%)
Drawdown (%)
Date

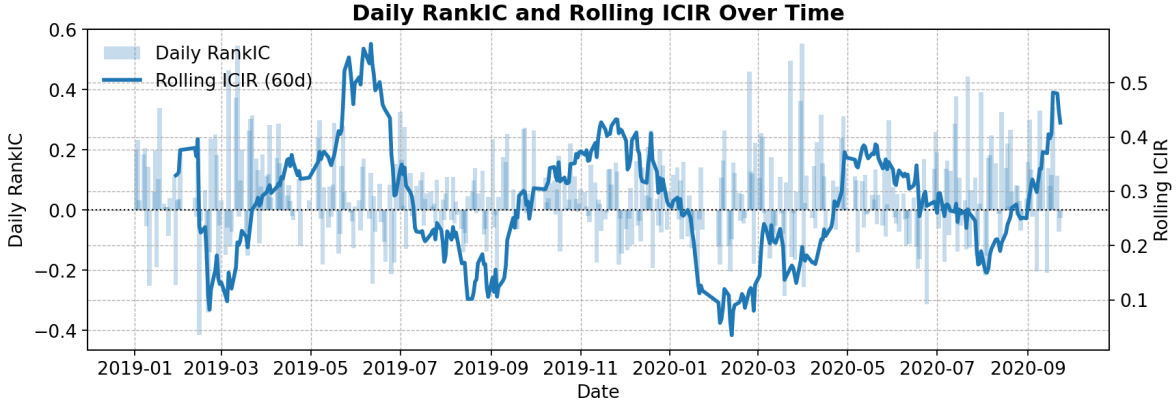

Long–Short Decile 10%
Cumulative P&L | Sharpe=4.08
Cumulative P&L
Date

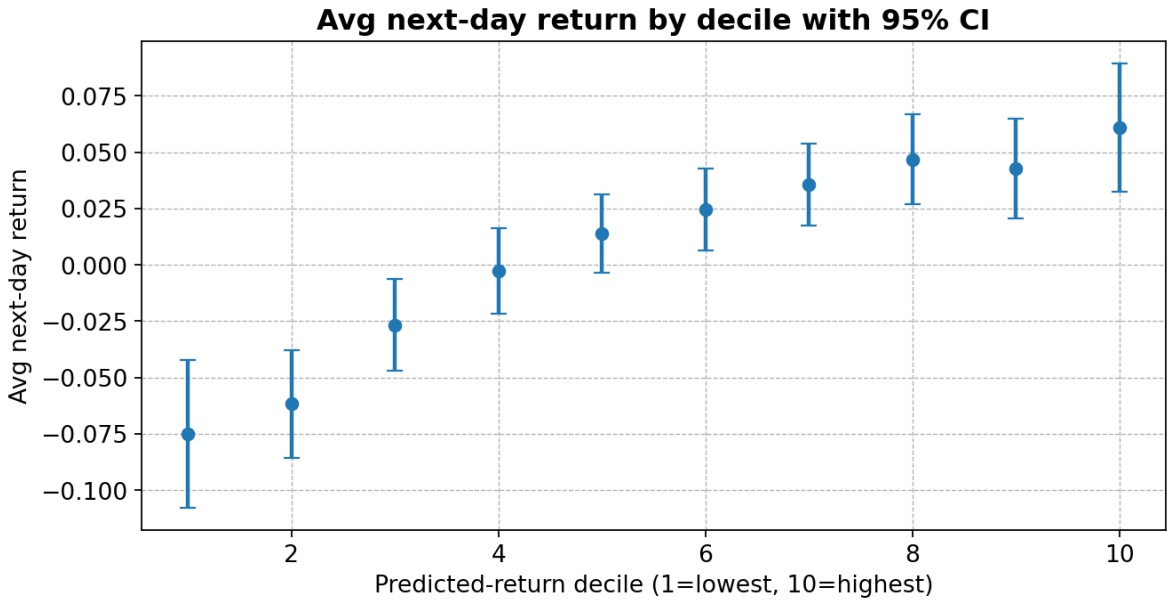

Long–Short Decile 10% — Drawdown (%)
Drawdown (%)
Date


Long-Short Decile 10% (Uncertainty-Penalized)
Cumulative P&L | Sharpe=4.03
Cumulative P&L
Date


Long-Short Decile 10% (Uncertainty-Penalized) — Drawdown (%)
Drawdown (%)
Date


Daily Cross-Sectional RankIC Over Time
Daily RankIC
60d Rolling Mean
Spearman RankIC
Date


Daily RankIC and Rolling ICIR Over Time
Daily RankIC
Rolling ICIR (60d)
Daily RankIC
Rolling ICIR
Date


Avg next-day return by decile with 95% CI
Avg next-day return
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
2
4
6
8
10
Predicted-return decile (1=lowest, 10=highest)