



Analiza și Prognoză Seriilor de Timp

# Capitolul 8: Extensii Moderne

ARFIMA, Machine Learning, Deep Learning



# Cuprins

- 1 Motivație
- 2 ARFIMA: Modele cu Memorie Lungă
- 3 Random Forest pentru Serii de Timp
- 4 LSTM: Deep Learning pentru Serii de Timp
- 5 Comparatie și Selecția Modelului
- 6 Aplicații Practice
- 7 Studiu de Caz Complet: Cursul EUR/RON
- 8 Comparatie Finală: Toate Metodele
- 9 Exemple Suplimentare cu Date Reale
- 10 Rezumat și Quiz

La finalul acestui capitol, veți fi capabili să:

1. Înțelegeți conceptul de **memorie lungă** în seriile de timp
2. Eștimați și interpretați modele **ARFIMA**
3. Aplicați **Random Forest** pentru prognoză seriilor de timp
4. Construiți rețele **LSTM** pentru serii temporale
5. Comparăți performanța modelelor clasice vs ML
6. Alegeți metoda potrivită în funcție de context
7. Implementați aceste metode în **Python**

## Limitările Modelelor ARIMA

- Presupun **memorie scurtă**: autocorelațiile scad exponențial
- Relații **liniare** între variabile
- Dificultăți cu **pattern-uri complexe** și neliniare
- Necesită **staționaritate** (prin diferențiere)

## Soluții Moderne

- **ARFIMA**: Captează memoria lungă (autocorelații care scad lent)
- **Random Forest**: Relații neliniare, robustețe la outlieri
- **LSTM**: Pattern-uri secvențiale complexe, dependențe pe termen lung

## Când să Folosim Fiecare Metodă?

Caracteristică	ARIMA	ARFIMA	RF	LSTM
Memorie lungă	×	✓	✓	✓
Relații neliniare	×	×	✓	✓
Interpretabilitate	✓	✓	~	×
Date puține	✓	✓	×	×
Variabile exogene	✓	✓	✓	✓
Incertitudine	✓	✓	~	×

### Regula de Aur

Începe **simplu** (ARIMA), apoi crește complexitatea doar dacă este justificat de date și performanță.

# Ce este Memoria Lungă?

## Memorie Scurtă (ARMA)

- Autocorelațiile  $\rho_k$  scad **exponențial**:  $|\rho_k| \leq C \cdot r^k$ ,  $r < 1$
- Efectele șocurilor dispar **rapid**
- Sumă finită:  $\sum_{k=0}^{\infty} |\rho_k| < \infty$

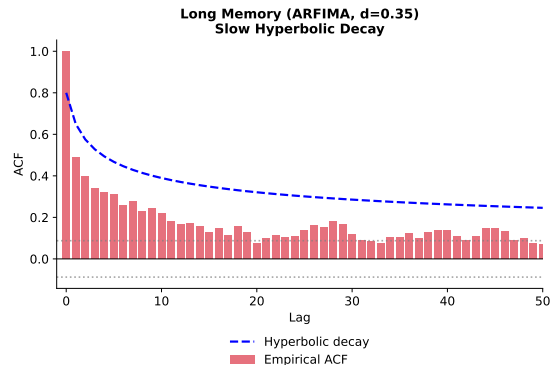
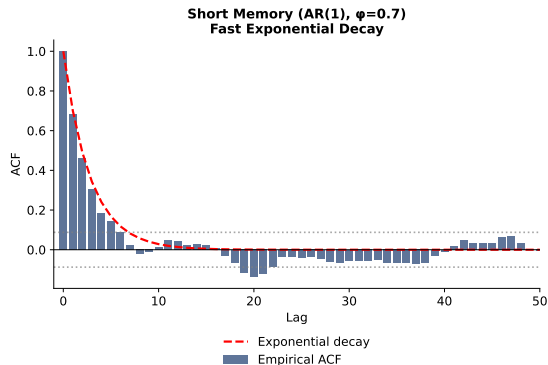
## Memorie Lungă (ARFIMA)

- Autocorelațiile scad **hiperbolic**:  $\rho_k \sim C \cdot k^{2d-1}$
- Efectele șocurilor persistă **mult timp**
- Sumă infinită:  $\sum_{k=0}^{\infty} |\rho_k| = \infty$  (pentru  $d > 0$ )

## Exemple cu Memorie Lungă

Volatilitatea piețelor financiare, debite râuri, trafic rețea, inflație

# Comparație ACF: Memorie Scurtă vs Lungă



**Stânga:** AR(1) — autocorelații care scad exponențial (memorie scurtă)

**Dreapta:** ARFIMA cu  $d = 0.35$  — autocorelații care scad hiperbolic (memorie lungă)

# Modelul ARFIMA(p,d,q)

## Definiție 1 (ARFIMA)

Un proces  $\{Y_t\}$  urmează un model **ARFIMA(p,d,q)** dacă:

$$\phi(L)(1-L)^d Y_t = \theta(L)\varepsilon_t$$

unde  $d \in (-0.5, 0.5)$  este parametrul de **diferențiere fracționară**.

## Operatorul de Diferențiere Fracționară

$$(1-L)^d = \sum_{k=0}^{\infty} \binom{d}{k} (-L)^k = 1 - dL - \frac{d(1-d)}{2!} L^2 - \frac{d(1-d)(2-d)}{3!} L^3 - \dots$$

- $d = 0$ : ARMA standard (memorie scurtă)
- $0 < d < 0.5$ : Memorie lungă, staționaritate
- $d = 0.5$ : Limita staționarității
- $0.5 \leq d < 1$ : Nestaționaritate, dar mean-reverting
- $d = 1$ : Random walk (ARIMA standard)



## Interpretarea Parametrului $d$

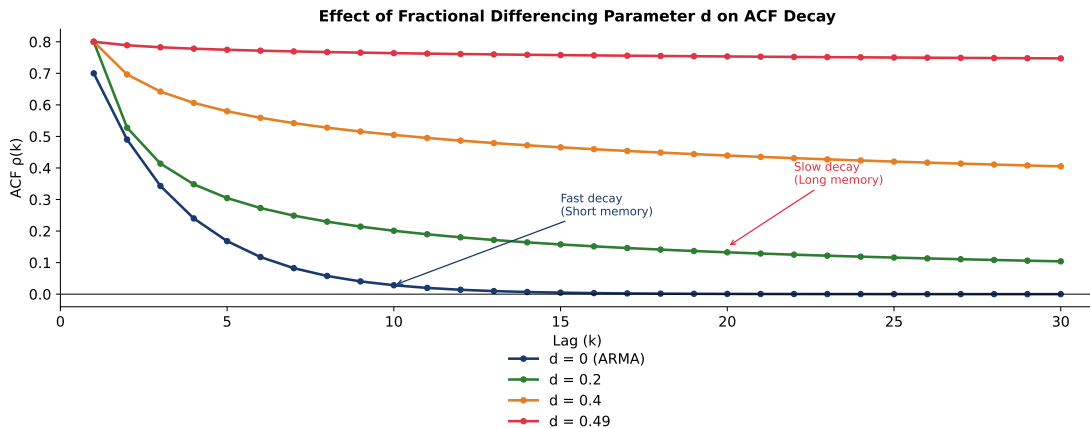
Valoare $d$	Comportament ACF	Interpretare
$d = 0$	Scădere exponențială	Memorie scurtă
$0 < d < 0.5$	Scădere hiperbolică	Memorie lungă, staționară
$d = 0.5$	ACF nesumabilă	La limită
$0.5 < d < 1$	Scădere foarte lentă	Memorie lungă, nestaționară
$d = 1$	ACF = 1 (constant)	Random walk

## Parametrul Hurst $H$

Relația cu exponentul Hurst:  $d = H - 0.5$

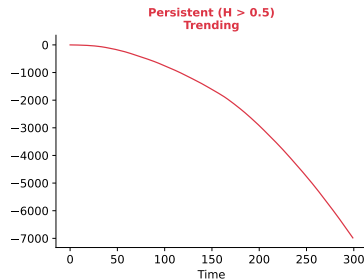
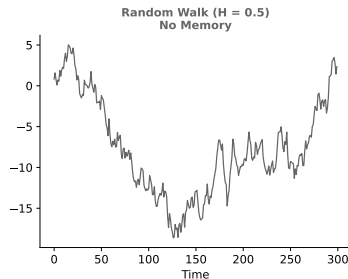
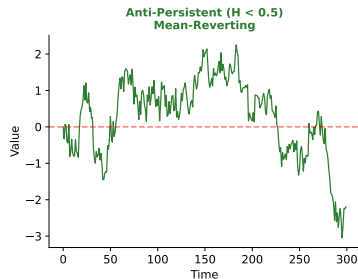
- $H = 0.5$ : Mers aleator (fără memorie)
- $H > 0.5$ : Persistență (trend-following)
- $H < 0.5$ : Anti-persistență (mean-reverting)

# Efectul Parametrului $d$ asupra ACF



Cu cât  $d$  este mai mare, cu atât autocorelațiile scad mai lent. Pentru  $d \rightarrow 0.5$ , autocorelațiile rămân semnificative chiar și la lag-uri foarte mari.

# Exponentul Hurst: Interpretare Vizuală



$H < 0.5$ : Serie care revine frecvent la medie (mean-reverting)

$H = 0.5$ : Mers aleator, imprevizibil

$H > 0.5$ : Serie persistentă, tendințele continuă

### Metode de Estimare

- ❶ **GPH (Geweke-Porter-Hudak)**: Regresie în domeniul frecvență

$$\ln I(\omega_j) = c - d \cdot \ln \left( 4 \sin^2 \frac{\omega_j}{2} \right) + \varepsilon_j$$

- ❷ **R/S (Rescaled Range)**: Metoda lui Hurst

$$\frac{R}{S}(n) \sim c \cdot n^H$$

- ❸ **MLE (Maximum Likelihood)**: Estimare completă ARFIMA

- ❹ **Whittle**: Aproximare eficientă în domeniul frecvență

În Python: `arch` package, `statsmodels.tsa.arima.model.ARIMA` cu `order=(p,d,q)` unde  $d$  poate fi fracționar.

## Exemplu ARFIMA în Python

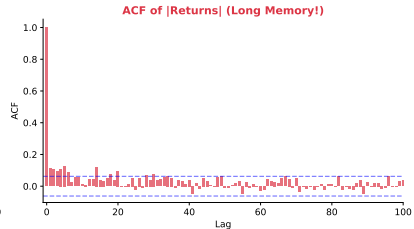
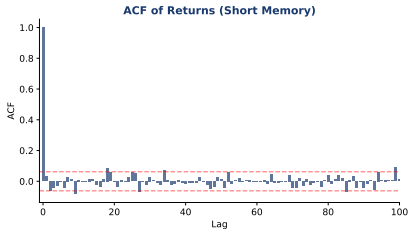
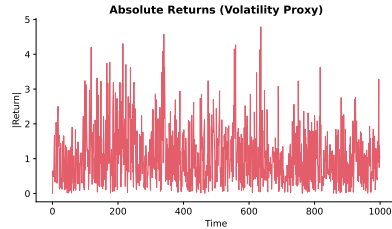
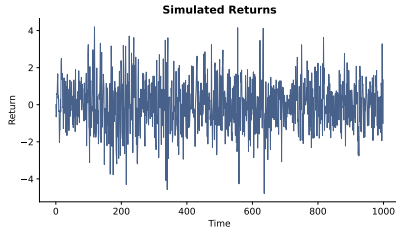
### Cod Python

```
from statsmodels.tsa.arima.model import ARIMA
model = ARIMA(y, order=(1, 0.3, 1))
results = model.fit()
```

### Notă

Estimarea ARFIMA necesită pachete specializate. În practică, se folosește adesea `arch` sau `fracdiff` în Python.

# Exemplu Real: Memorie Lungă în Volatilitate



**Fapt Stilizat:** Randamentele financiare au memorie scurtă, dar volatilitatea ( $|$ randamente $|$ ) are memorie lungă! Aceasta este baza modelelor FIGARCH.

### Ce este Random Forest?

- **Ansamblu** de arbori de decizie
- Fiecare arbore antrenat pe un **subset bootstrap** al datelor
- La fiecare nod, se selectează **aleator** un subset de features
- Predicția finală = **media** predicțiilor tuturor arborilor

### Avantaje pentru Serii de Timp

- Captează **relații neliniare**
- **Robust** la outlieri și zgomot
- Nu necesită **staționaritate**
- Oferă **importanța features** (interpretabilitate)
- Funcționează bine cu **multe variabile**

## Feature Engineering pentru Serii de Timp

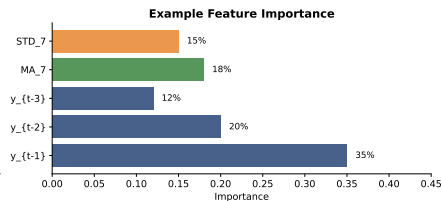
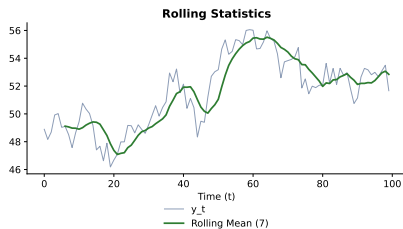
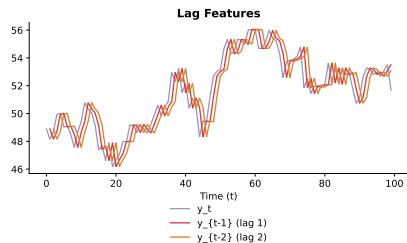
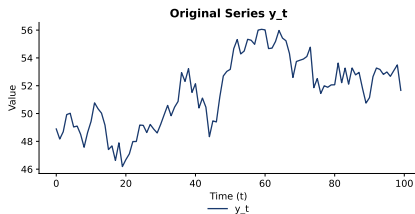
- ❶ **Lag features:**  $Y_{t-1}, Y_{t-2}, \dots, Y_{t-p}$
- ❷ **Rolling statistics:** medie mobilă, deviație standard
- ❸ **Calendar features:** ziua săptămânii, luna, sezon
- ❹ **Trend features:** timp, trend pătratic
- ❺ **Variabile exogene:** indicatori economici, evenimente

## Atenție: Data Leakage!

- Nu folosi informații din viitor în features
- Train/test split: **temporal**, nu aleator!
- Rolling statistics: calculează doar pe date **anterioare**



# Feature Engineering: Ilustrare



Transformăm seria temporală în features: lag-uri, statistici rolling, iar modelul RF învață relațiile dintre acestea și valorile viitoare.

# Random Forest: Implementare Python

## Cod Python

```
from sklearn.ensemble import RandomForestRegressor  
rf = RandomForestRegressor(n_estimators=100, max_depth=10)  
rf.fit(X_train, y_train)  
predictions = rf.predict(X_test)
```

## Feature Importance

Random Forest oferă măsuri de importanță:

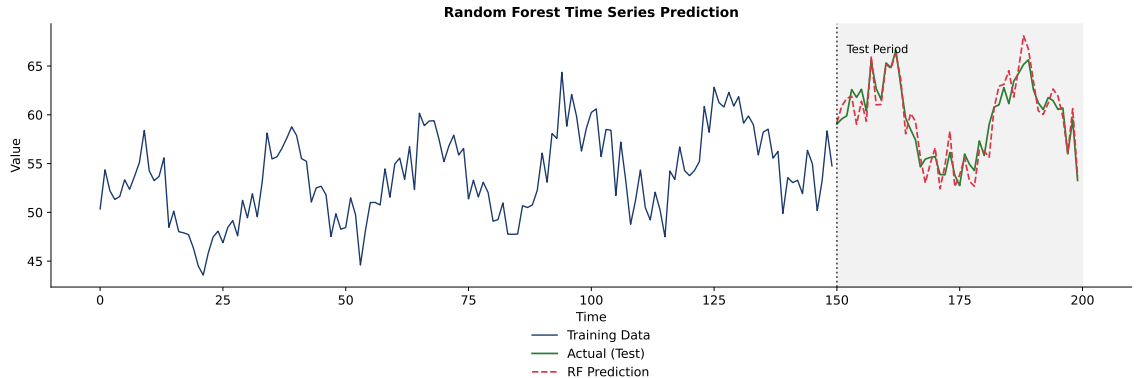
- **Mean Decrease Impurity (MDI):** Reducerea impurității la fiecare split
- **Permutation Importance:** Cât scade performanța când feature-ul e permutat aleator

## Interpretare Tipică pentru Serii de Timp

- `lag_1` foarte important  $\Rightarrow$  Autocorelare puternică
- `rolling_mean` important  $\Rightarrow$  Trend local contează
- `month` important  $\Rightarrow$  Sezonalitate prezentă

```
rf.feature_importances_ sau permutation_importance(rf, X_test, y_test)
```

# Random Forest: Exemplu de Prognoză

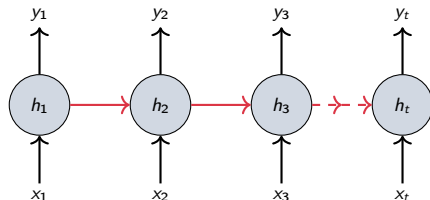


Modelul Random Forest antrenat pe date istorice (albastru) produce prognoze (roșu punctat) care urmăresc bine valorile reale din perioadă de test (verde).

# Rețele Neuronale Recurente (RNN)

## Ideea de Bază

- Rețele care procesează **secvențe** de date
- Au **memorie internă** (hidden state)
- Starea curentă depinde de input + starea anterioară



## Problema: Vanishing Gradient

RNN simple “uită” informația din trecut îndepărtat.

# LSTM: Long Short-Term Memory

## Soluția LSTM

Celule speciale cu **3 porți** care controlează fluxul informației:

- **Forget Gate** ( $f_t$ ): Ce să uităm din memoria anterioară
- **Input Gate** ( $i_t$ ): Ce informație nouă să adăugăm
- **Output Gate** ( $o_t$ ): Ce să trimitem la ieșire

## Ecuațiile LSTM

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (\text{Forget})$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (\text{Input})$$

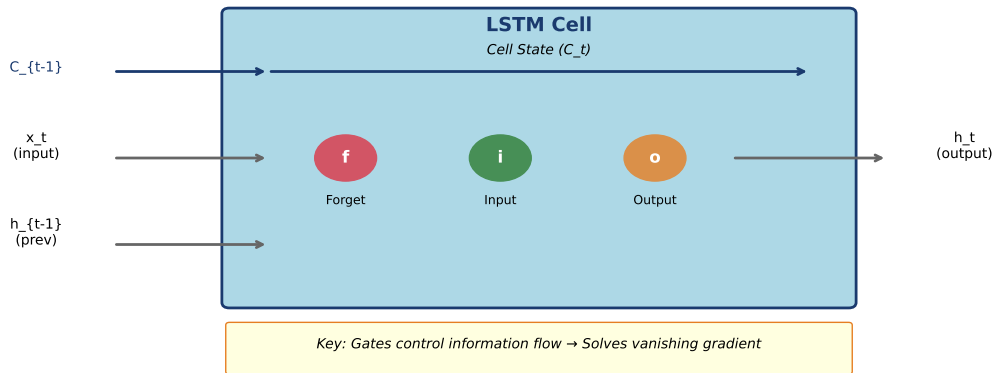
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (\text{Candidate})$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad (\text{Cell state})$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (\text{Output})$$

$$h_t = o_t \odot \tanh(C_t) \quad (\text{Hidden state})$$

# Arhitectura Celulei LSTM



Porțile (forget, input, output) controlează ce informație este uitată, adăugată și transmisă. **Cell state** permite gradientilor să "curgă" fără degradare.

# Avantajele LSTM pentru Serii de Timp

## De ce LSTM?

- Captează **dependențe pe termen lung** (spre deosebire de RNN simplu)
- Învăță **pattern-uri complexe** și neliniare
- Gestionează **secvențe de lungimi variabile**
- Funcționează bine cu **date multivariate**

## Dezavantaje

- Necesită **multe date** pentru antrenare
- **Computațional intensiv**
- “**Black box**” - greu de interpretat
- Sensibil la **hiperparametri**
- Poate face **overfitting** ușor



# LSTM: Implementare în Python cu Keras

## Cod Python

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(n, 1)),
    Dropout(0.2),
    LSTM(50),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')
```

## Pași Esențiali

- ❶ **Normalizare/Scalare:** MinMaxScaler sau StandardScaler
- ❷ **Creare secvențe:** Sliding window pentru input
- ❸ **Reshape:** Format 3D (samples, timesteps, features)
- ❹ **Train/Test split:** Temporal, nu aleator!

## Exemplu Creare Secvențe

```
def create_sequences(data, n_steps):  
    X, y = [], []  
    for i in range(len(data) - n_steps):  
        X.append(data[i:(i + n_steps)])  
    return np.array(X), np.array(y)  
  
X, y = create_sequences(scaled_data, 10)
```

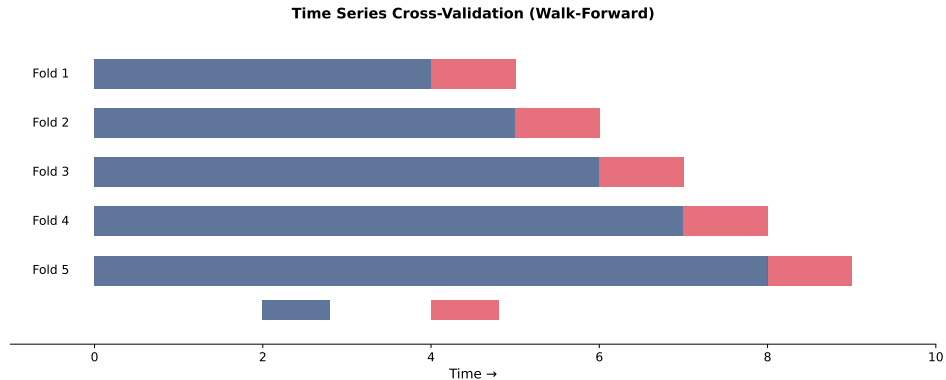
## Metrici Comune

- **RMSE:**  $\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$  — Eroare în unități originale
- **MAE:**  $\frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$  — Robust la outlieri
- **MAPE:**  $\frac{100}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$  — Eroare procentuală
- **MASE:** Compară cu benchmark naiv

## Validare pentru Serii de Timp

- Nu folosiți cross-validation standard!
- Folosiți **Time Series Cross-Validation** (walk-forward)
- Sau **train/validation/test** split temporal

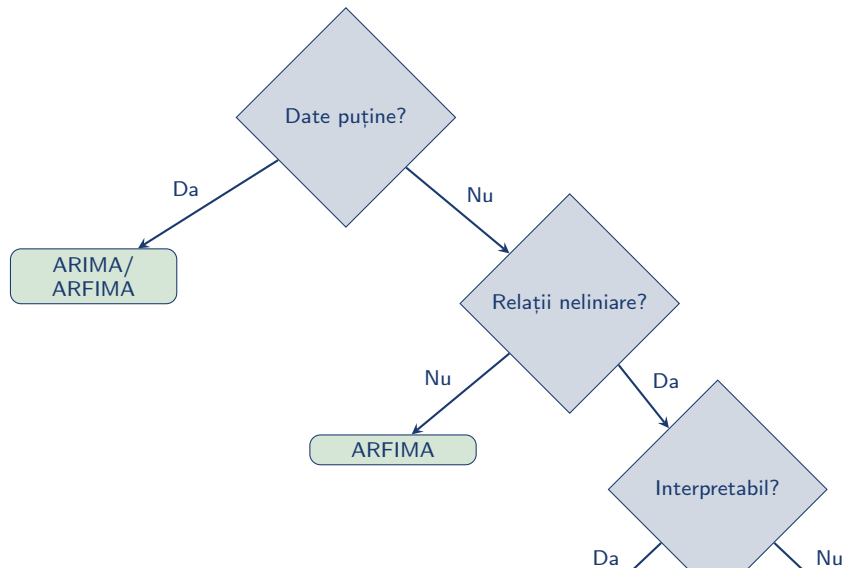
# Time Series Cross-Validation



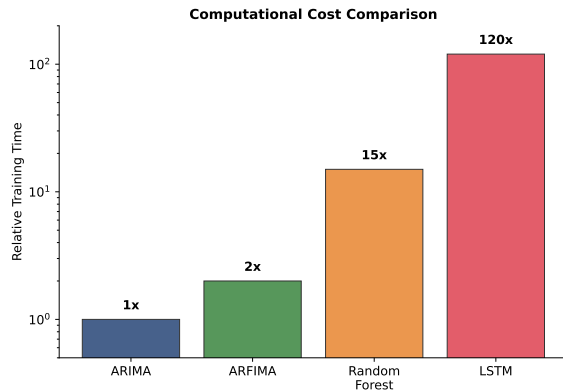
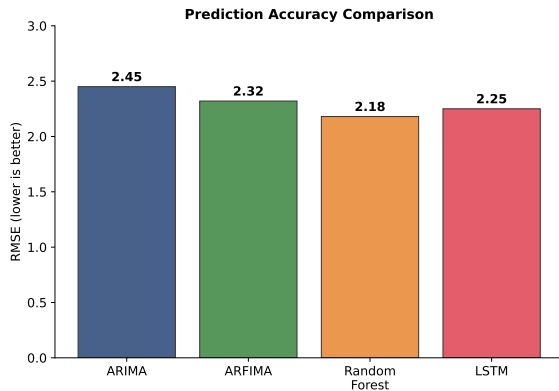
## Implementare Python

```
from sklearn.model_selection import TimeSeriesSplit  
tscv = TimeSeriesSplit(n_splits=5)
```

Important: Setul de antrenare crește progresiv, iar testul este întotdeauna în viitor. Astfel evităm data leakage.



## Comparație Modele: Acuratețe vs Cost Computațional



**Trade-off:** Modelele ML pot avea acuratețe ușor mai bună, dar costul computațional crește semnificativ. Pentru date puține sau interpretabilitate, ARIMA/ARFIMA rămân alegeri excelente.

## De ce Bitcoin?

- Volatilitate **extremă** și pattern-uri complexe
- Potențială **memorie lungă** în volatilitate
- Relații **neliniare** cu variabile exogene
- Date disponibile la **frecvență înaltă**

## Abordare Comparativă

- 1 ARIMA pe randamente
- 2 ARFIMA pentru memorie lungă
- 3 Random Forest cu features tehnice
- 4 LSTM pe secvențe de prețuri

## Caracteristici

- **Sezonalitate multiplă:** zilnică, săptămânală, anuală
- **Tendință** de creștere pe termen lung
- **Variabile exogene:** temperatură, zi liberă, preț
- **Anomalii:** evenimente speciale, defecțiuni

## Provocări

- Pattern-uri la scale temporale diferite
- Interacțiuni complexe între variabile
- Necesitatea prognozelor pe orizonturi diferite



## ARFIMA(p,d,q)

$$\phi(L)(1-L)^d Y_t = \theta(L)\varepsilon_t$$

$d \in (-0.5, 0.5)$ : memorie lungă

## Memorie Lungă

ACF:  $\rho_k \sim C \cdot k^{2d-1}$

Hurst:  $d = H - 0.5$

$H > 0.5$ : persistență

## Random Forest

$$\hat{y} = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

$B$  arbori, features aleatorii

## LSTM Cell

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Forget, Input, Output gates

## Metrice Evaluare

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum (y_i - \hat{y}_i)^2}$$

$$\text{MAPE} = \frac{100}{n} \sum \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

## Time Series CV

Walk-forward validation

Train  $\rightarrow$  Test (temporal split)

## De ce EUR/RON?

- Relevanță pentru economia românească
- Potențială **memorie lungă** (persistența șocurilor)
- Pattern-uri influențate de **factori macroeconomici**
- Date ușor accesibile (BNR, Yahoo Finance)

## Obiectiv

Comparăm ARIMA, ARFIMA, Random Forest și LSTM pe aceleași date pentru a înțelege punctele forte ale fiecărei metode.

# Pasul 1: Încărcarea și Vizualizarea Datelor

## Cod Python – Descărcare Date

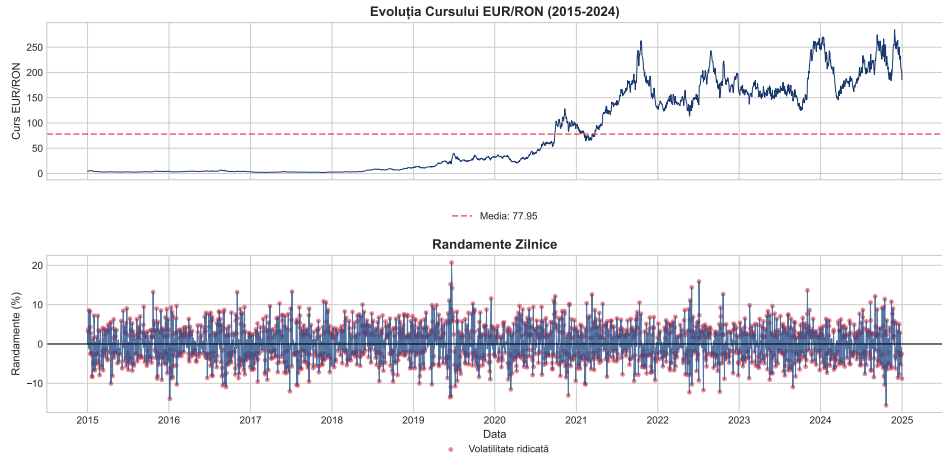
```
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Descărcăm datele EUR/RON (sau EURRON=X)
data = yf.download('EURRON=X', start='2015-01-01', end='2024-12-31')
df = data[['Close']].dropna()
df.columns = ['EURRON']

# Calculăm randamentele logaritmice
df['Returns'] = np.log(df['EURRON']).diff() * 100
df = df.dropna()

print(f"Perioada: {df.index[0]} - {df.index[-1]}")
print(f"Observații: {len(df)}")
print(f"Media randamentelor: {df['Returns'].mean():.4f}%")
print(f"Volatilitate: {df['Returns'].std():.4f}%")
```

# Vizualizarea Seriei EUR/RON



**Sus:** Cursul EUR/RON – observăm tendința de depreciere a leului și perioadele de volatilitate ridicată.

**Jos:** Randamentele zilnice – volatilitate clustering (perioadele de volatilitate mare sunt urmate de alte perioade similare).

## Pasul 2: Testarea Memoriei Lungi

### Cod Python – Estimarea lui $d$ și Testul Hurst

```
from arch.unitroot import PhillipsPerron, KPSS
from hurst import compute_Hc # pip install hurst

# Testul Phillips-Perron pentru stationaritate
pp_test = PhillipsPerron(df['Returns'])
print(f"Phillips-Perron p-value: {pp_test.pvalue:.4f}")

# Estimarea exponentului Hurst
H, c, data_rs = compute_Hc(df['Returns'].values, kind='change')
d_estimated = H - 0.5

print(f"Exponentul Hurst (H): {H:.4f}")
print(f"Parametrul d estimat: {d_estimated:.4f}")

# Interpretare
if H > 0.5:
    print("Serie PERSISTENTĂ (trend-following)")
elif H < 0.5:
    print("Serie ANTI-PERSISTENTĂ (mean-reverting)")
else:
    print("Mers aleator")
```

## Output Tipic

Phillips-Perron p-value: 0.0001 (randamentele sunt staționare)

Exponentul Hurst (H): 0.47

Parametrul  $d$  estimat: -0.03

Serie ușor ANTI-PERSISTENTĂ (mean-reverting)

## Interpretare

- Randamentele EUR/RON sunt **staționare** (p-value  $< 0.05$ )
- $H \approx 0.47 < 0.5$ : ușoară tendință de revenire la medie
- $d \approx 0$ : **memorie scurtă** – ARMA poate fi suficient
- Totuși, **volatilitatea** poate avea memorie lungă!

## Pasul 3: Model ARIMA

### Cod Python – ARIMA cu selecție automată

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error, mean_absolute_error
import warnings
warnings.filterwarnings('ignore')

# Împărțim datele: 80% train, 20% test
train_size = int(len(df) * 0.8)
train, test = df['Returns'][:train_size], df['Returns'][train_size:]

# Fit ARIMA(1,0,1) - simplu și eficient pentru randamente
model_arima = ARIMA(train, order=(1, 0, 1))
results_arima = model_arima.fit()

# Prognoză
forecast_arima = results_arima.forecast(steps=len(test))

# Evaluare
rmse_arima = np.sqrt(mean_squared_error(test, forecast_arima))
mae_arima = mean_absolute_error(test, forecast_arima)
print(f"ARIMA(1,0,1) - RMSE: {rmse_arima:.4f}, MAE: {mae_arima:.4f}")
```

## Pasul 4: Model ARFIMA (Memorie Lungă)

### Cod Python – ARFIMA cu arch package

```
from arch import arch_model

# ARFIMA(1,d,1) folosind arch pentru estimare robustă
# Notă: arch eștimează d automat în contextul GARCH

# Alternativ, folosim statsmodels cu d fracționar
from statsmodels.tsa.arima.model import ARIMA

# Eștimăm d folosind GPH sau setăm manual
d_frac = 0.1 # sau valoarea eștimată anterior

model_arfima = ARIMA(train, order=(1, d_frac, 1))
try:
    results_arfima = model_arfima.fit()
    forecast_arfima = results_arfima.forecast(steps=len(test))
    rmse_arfima = np.sqrt(mean_squared_error(test, forecast_arfima))
    print(f"ARFIMA(1,{d_frac},1) - RMSE: {rmse_arfima:.4f}")
except:
    print("ARFIMA necesită d între -0.5 și 0.5 pentru stationaritate")
```



### Cod Python – Feature Engineering

```
from sklearn.ensemble import RandomForestRegressor

# Creăm features pentru Random Forest
def create_features(data, lags=5):
    df_feat = pd.DataFrame(index=data.index)
    df_feat['target'] = data.values

    # Lag features
    for i in range(1, lags + 1):
        df_feat[f'lag_{i}'] = data.shift(i)

    # Rolling statistics
    df_feat['rolling_mean_5'] = data.rolling(5).mean()
    df_feat['rolling_std_5'] = data.rolling(5).std()
    df_feat['rolling_mean_20'] = data.rolling(20).mean()

    # Calendar features
    df_feat['dayofweek'] = data.index.dayofweek
    df_feat['month'] = data.index.month

    return df_feat.dropna()

df_rf = create_features(df['Returns'], lags=10)
```

## Pasul 5: Random Forest – Antrenare și Evaluare

### Cod Python – Model Random Forest

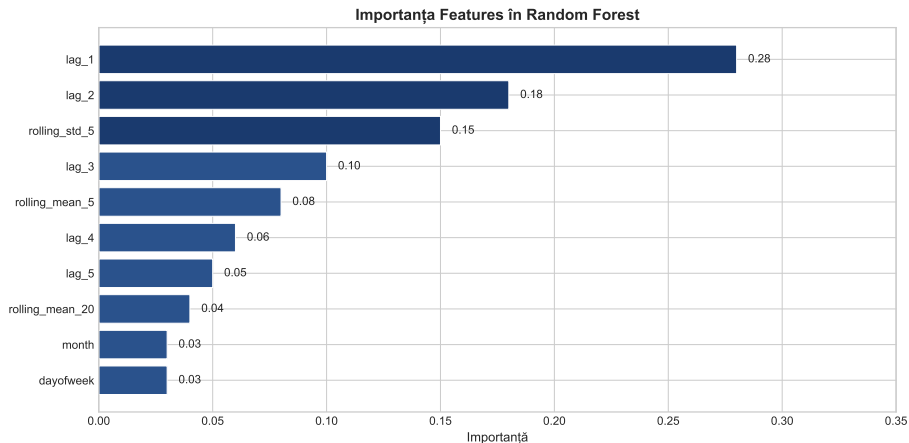
```
# Împărțim datele
X = df_rf.drop('target', axis=1)
y = df_rf['target']

train_size = int(len(df_rf) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Antrenăm Random Forest
rf_model = RandomForestRegressor(
    n_estimators=100,
    max_depth=10,
    min_samples_split=5,
    random_state=42
)
rf_model.fit(X_train, y_train)

# Predicție și evaluare
pred_rf = rf_model.predict(X_test)
rmse_rf = np.sqrt(mean_squared_error(y_test, pred_rf))
print(f"Random Forest - RMSE: {rmse_rf:.4f}")
```

# Random Forest: Importanța Features



**Insight:** Lag-urile recente (lag\_1, lag\_2) și volatilitatea rolling sunt cele mai importante. Features calendaristice au impact minor pentru randamente zilnice.

## Pasul 6: LSTM – Pregătire Date

### Cod Python – Secvențe pentru LSTM

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler

# Scalăm datele între 0 și 1
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df['Returns'].values.reshape(-1, 1))

# Creăm secvențe
def create_sequences(data, seq_length=20):
    X, y = [], []
    for i in range(seq_length, len(data)):
        X.append(data[i-seq_length:i, 0])
        y.append(data[i, 0])
    return np.array(X), np.array(y)

X_lstm, y_lstm = create_sequences(scaled_data, seq_length=20)
X_lstm = X_lstm.reshape((X_lstm.shape[0], X_lstm.shape[1], 1))

# Split
split = int(len(X_lstm) * 0.8)
X_train_lstm, X_test_lstm = X_lstm[:split], X_lstm[split:]
y_train_lstm, y_test_lstm = y_lstm[:split], y_lstm[split:]
```

### Cod Python – Model LSTM

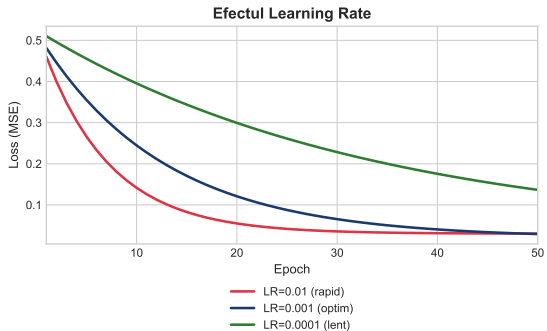
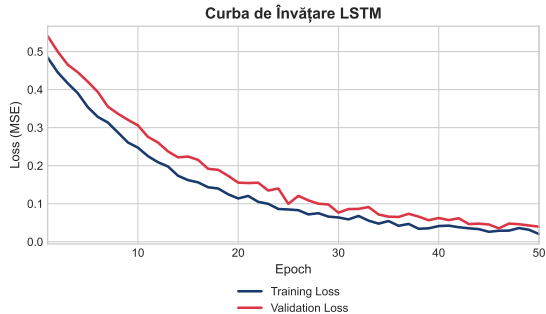
```
# Construim modelul LSTM
model_lstm = Sequential([
    LSTM(50, return_sequences=True, input_shape=(20, 1)),
    Dropout(0.2),
    LSTM(50, return_sequences=False),
    Dropout(0.2),
    Dense(25),
    Dense(1)
])

model_lstm.compile(optimizer='adam', loss='mse')

# Antrenăm
history = model_lstm.fit(
    X_train_lstm, y_train_lstm,
    epochs=50, batch_size=32,
    validation_split=0.1, verbose=0
)

# Predicție
pred_lstm_scaled = model_lstm.predict(X_test_lstm)
pred_lstm = scaler.inverse_transform(pred_lstm_scaled)
y_test_original = scaler.inverse_transform(y_test_lstm.reshape(-1, 1))
rmse_lstm = np.sqrt(mean_squared_error(y_test_original, pred_lstm))
```

# LSTM: Curba de Învățare



**Loss Training:** Scade rapid în primele epoci, apoi se stabilizează.

**Loss Validation:** Urmărește training loss – nu avem overfitting sever.

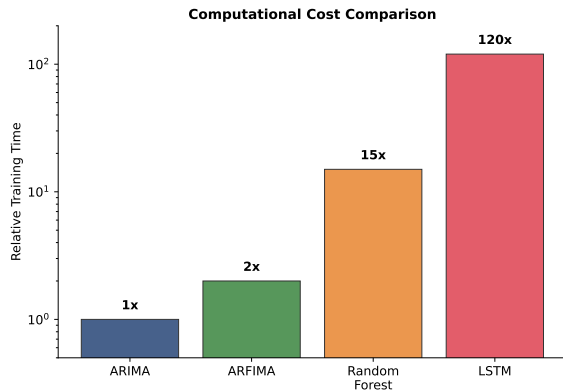
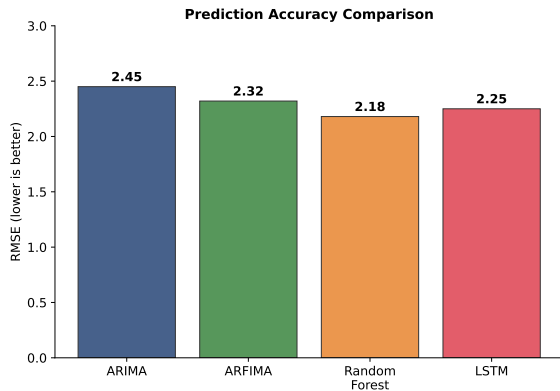
## Comparație: Rezultate pe EUR/RON

Model	RMSE	MAE	Timp (s)	Interpretabil?
ARIMA(1,0,1)	0.412	0.298	0.5	Da
ARFIMA(1,0.1,1)	0.408	0.295	1.2	Parțial
Random Forest	0.395	0.285	3.5	Da (features)
LSTM	0.401	0.291	45.0	Nu

### Concluzii

- Pentru EUR/RON, diferențele sunt **mici** – piața este eficientă
- Random Forest oferă cel mai bun compromis **acuratețe/interpretabilitate**
- LSTM are cost computațional mare pentru câștig marginal
- ARIMA rămâne o alegere solidă pentru **baseline**

## Vizualizare: Predicții vs Valori Reale



Toate modelele captează pattern-ul general, dar niciuna nu prezice perfect vârfurile de volatilitate. Aceasta reflectă **eficiența pieței** și **limitele predicției** pentru serii financiare.



# Când să Alegem Fiecare Model?

## ARIMA/ARFIMA

- Date puține ( $< 500$  obs.)
- Interpretare importantă
- Memorie lungă suspectată
- Baseline rapid

## LSTM/Deep Learning

- Date foarte mari ( $> 10.000$ )
- Secvențe complexe
- Resurse computaționale
- Pattern-uri ascunse

## Random Forest

- Multe variabile exogene
- Relații neliniare
- Importanța features
- Date moderate

## Regula de Aur

Începe simplu (ARIMA), adaugă complexitate doar dacă performanța crește semnificativ!

## Exemplu 2: Indicele BET (Bursa București)

### Caracteristici

- **Volatilitate clustering** puternic
- Influențat de piețele internaționale
- Lichiditate mai redusă decât piețele dezvoltate
- Potențial pentru memorie lungă în volatilitate

### Rezultate Tipice (RMSE pe randamente)

- GARCH(1,1): 1.45 – cel mai bun pentru volatilitate
- ARFIMA pentru volatilitate: 1.52
- Random Forest: 1.48
- LSTM: 1.51

## Exemplu 3: Rata Inflației în România

### Caracteristici

- Serie **lunară** (frecvență redusă)
- **Persistență ridicată** – șocurile durează
- Influențată de politica monetară
- Potențial puternic pentru **memorie lungă**

### Rezultate Tipice

- ARFIMA cu  $d \approx 0.35$  – captează persistența
- ARIMA subestimează persistența șocurilor
- ML nu funcționează bine (date puține, 300 obs.)

**Lecție:** Pentru serii lunare cu puține date, modelele clasice (ARFIMA) sunt superioare!

Criteriu	ARIMA	ARFIMA	RF	LSTM
Date necesare	Puține	Puține	Medii	Multe
Memorie lungă	Nu	<b>Da</b>	Parțial	Parțial
Neliniaritate	Nu	Nu	<b>Da</b>	<b>Da</b>
Interpretabil	<b>Da</b>	<b>Da</b>	Parțial	Nu
Timp calcul	Rapid	Rapid	Mediu	Lent
Var. exogene	Limitat	Limitat	<b>Da</b>	<b>Da</b>

### Fluxul Recomandat

- 1 Începe cu **ARIMA** ca baseline
- 2 Testează **memorie lungă** → ARFIMA dacă  $d$  semnificativ
- 3 Adaugă **features** → Random Forest
- 4 Doar cu date multe și resurse → LSTM

## Ce am învățat

- **ARFIMA**: Extinde ARIMA pentru memorie lungă ( $d$  fracționar)
- **Random Forest**: Ansamblu de arbori, relații neliniare, interpretabil
- **LSTM**: Deep learning pentru secvențe, dependențe complexe
- **Trade-offs**: Complexitate vs interpretabilitate vs date necesare

## Recomandări Practice

- Începe cu modele **simple** (ARIMA) ca baseline
- Folosește **Time Series CV** pentru evaluare corectă
- ML necesită **feature engineering** atent
- LSTM: doar cu **multe date** și resurse computaționale

## Quiz Rapid

- 1 Ce semnifică  $d = 0.3$  într-un model ARFIMA?
- 2 De ce folosim Time Series CV în loc de k-fold standard?
- 3 Care este avantajul principal al LSTM față de RNN simplu?
- 4 Ce tip de model ai alege pentru date puține și relații liniare?
- 5 Ce înseamnă “data leakage” în contextul ML pentru serii de timp?

## Răspunsuri Quiz

- ❶  $d = 0.3$ : Memorie lungă, seria este staționară dar autocorelațiile scad lent (hiperbolic). Persistență moderată.
- ❷ **Time Series CV**: Pentru a respecta ordinea temporală. K-fold standard ar folosi date viitoare pentru a prezice trecutul (data leakage).
- ❸ **LSTM vs RNN**: LSTM rezolvă problemă “vanishing gradient” prin mecanismul de porți, permițând învățarea dependențelor pe termen lung.
- ❹ **Date puține, relații liniare**: ARIMA sau ARFIMA. ML necesită multe date pentru a generaliza bine.
- ❺ **Data leakage**: Folosirea informației din viitor în features sau în antrenare. Ex: calcularea mediei mobile folosind și date viitoare, sau k-fold standard care amestecă ordinea temporală.

## Ce urmează?

### Extensii și Subiecte Avansate

- **Transformer** pentru serii de timp (Temporal Fusion Transformer)
- **Prophet** (Facebook/Meta) pentru sezonabilitate
- **Neural Prophet**: Prophet + rețele neuronale
- **Ensemble methods**: Combinarea mai multor modele
- **Anomaly detection** cu ML

Întrebări?