IMEN574/IDSC574 Programming for Data Science

22. **AI for Data Science**

Spring 2026

Instructor: **Sangwoo Mo**

Agenda



Part 1

AI for Tabular Data

- Why tabular data is hard for DL
- XGBoost / LightGBM: recap & code
- TabPFN: concept, architecture, code, benchmarks
- Text-to-SQL & Table QA



Part 2

AI for Time Series

- Prophet: decomposition, code, limits
- Foundation Models: TimesFM, Chronos, Moirai
- Practical guide: choosing the right approach

PART 1

AI for Tabular Data



Tabular Data Is Everywhere

Healthcare

Patient records, lab results, diagnoses — heterogeneous features

Finance

Transaction logs, credit scoring, portfolio risk — high stakes

E-commerce

User behavior, product attributes, click-through prediction

Manufacturing

Sensor readings, quality control, predictive maintenance

Marketing

Customer segmentation, churn prediction, A/B tests

Science

Genomics, materials discovery, clinical trial results

Why Is Tabular Data Challenging for Deep Learning?

Heterogeneous Features

Mixed types: categorical, numerical, ordinal — no spatial or sequential structure

Small Dataset Sizes

Many real-world tabular datasets have only hundreds to low thousands of rows

Feature Interactions

Complex non-linear interactions that tree-based models capture naturally via splits

No Universal Pre-training

Unlike vision (ImageNet) or NLP (text corpora), no standard pre-training existed

Deep Learning vs. Tree-Based Models on Tabular Data

Grinsztajn et al. (NeurIPS 2022): "Why do tree-based models still outperform deep learning on typical tabular data?"

Trees are robust to uninformative features

Trees preserve orientation of data (rotation invariance hurts NNs)

Trees learn irregular target functions easily

NNs tend to oversmooth and need more data

XGBoost: Quick Recap



Gradient Boosting

Sequentially train weak learners (trees), each correcting errors of the ensemble

Regularization

L1/L2 penalties on leaf weights + tree pruning to prevent overfitting

Column Subsampling

Random feature selection per tree/split — reduces correlation between trees

$$Obj(t) = \Sigma_i \; L(y_i, \; \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Loss on prediction + regularization on tree complexity

XGBoost: Code Example



```python
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load and split
X, y = load_breast_cancer(return_X_y=True)
X_tr, X_te, y_tr, y_te = train_test_split(
    X, y, test_size=0.2)

# Train with hyperparameters
model = xgb.XGBClassifier(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_alpha=0.1,
)
model.fit(X_tr, y_tr)

preds = model.predict(X_te)
print(f"Accuracy: {accuracy_score(y_te, preds):.4f}")
```

LightGBM: Key Differences from XGBoost

Feature	XGBoost	LightGBM
Split Strategy	Level-wise (breadth-first)	**Leaf-wise (best-first)**
Speed	Fast	**Faster (histogram-based)**
Memory	Higher	**Lower (binning)**
Categorical	Requires encoding	**Native support**
Overfitting Risk	**Lower**	Higher on small data

> *Both remain the gold standard for tabular prediction — can foundation models change this?*

LightGBM: Code Example



Note: num_leaves instead of max_depth — the key difference in practice

```
import lightgbm as lgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split




# LightGBM native Dataset format
train_set = lgb.Dataset(X_tr, label=y_tr)

params = {
    'objective': 'binary',
    'metric': 'binary_logloss',
    'num_leaves': 31,          # leaf-wise control
    'learning_rate': 0.05,
    'feature_fraction': 0.8,   # column sampling
    'verbose': -1
}

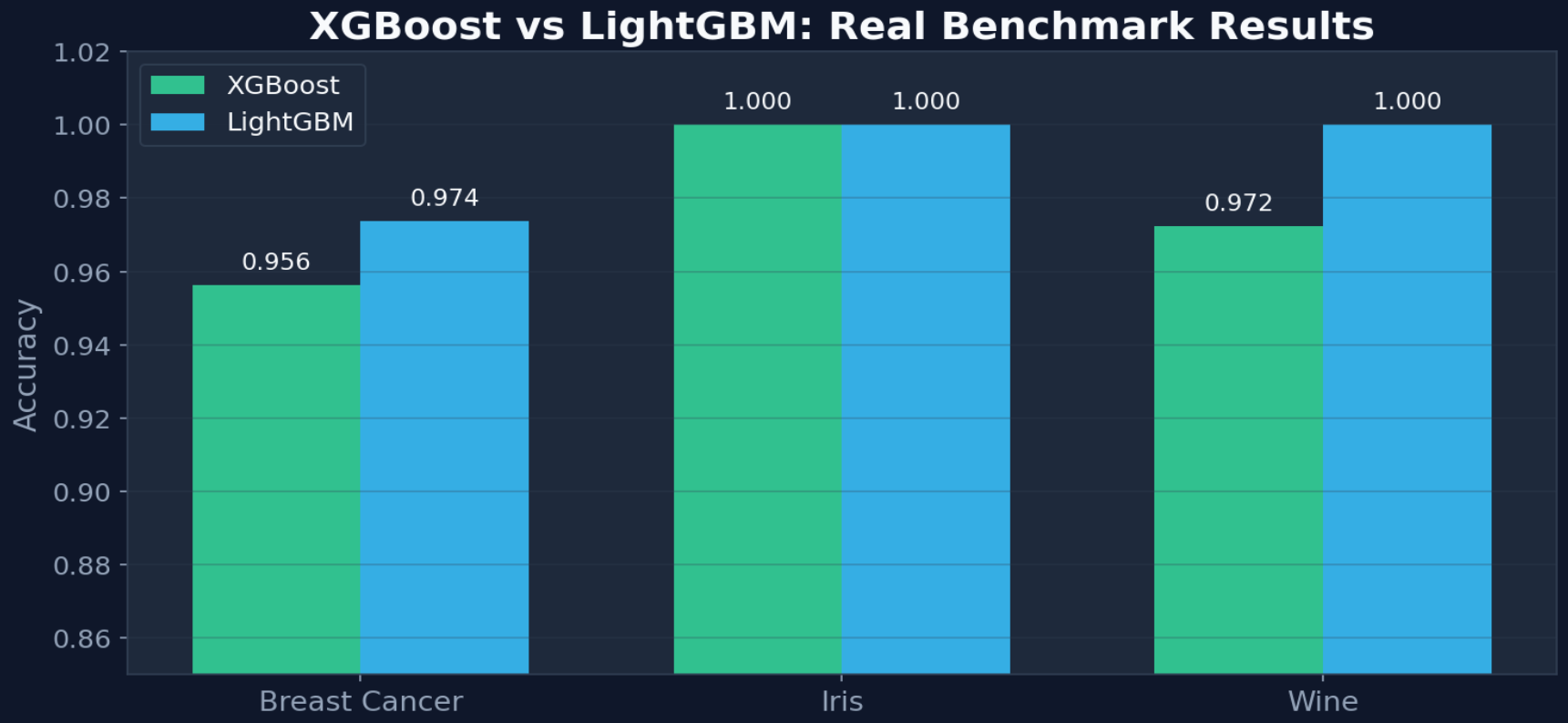
model = lgb.train(params, train_set, num_boost_round=100)
```

Why XGBoost / LightGBM Still Dominate

- Handle heterogeneous features natively (no preprocessing needed)
- Robust to irrelevant / noisy features — built-in feature selection via splits
- Excellent performance on small to large datasets (100 – 10M+ rows)
- Fast training & inference — production-ready with mature tooling
- Interpretable: SHAP values, feature importance, partial dependence plots
- Kaggle competitions: tree-based models win ~70% of tabular tasks

Demo: XGBoost vs LightGBM — Real Benchmark

Actual results on sklearn datasets — both achieve >95% accuracy across the board



Can We Build a Foundation Model for Tabular Data?

Vision

ImageNet → ResNet/ViT
→ fine-tune on any image task

NLP

Text corpora → GPT/BERT
→ fine-tune or prompt for any text task

Tabular?

No universal schema
→ ???
→ Train from scratch each time



TabPFN's Insight: Instead of pre-training on real tabular data (which differs across domains), pre-train on millions of synthetic datasets sampled from a flexible prior distribution.

TabPFN: Prior-Data Fitted Networks

Hollmann et al. (ICLR 2023)

1 Define Prior

Specify a prior P(D) over tabular datasets — structural causal models + Bayesian NNs

2 Pre-train

Train a Transformer on millions of synthetic datasets sampled from this prior

3 Inference

Feed real training data as context → predict test labels in a single forward pass

Key: TabPFN approximates Bayesian inference — it learns to be the optimal predictor given a prior over datasets.

TabPFN: Architecture



Training

- ~12M params (small by modern standards)
- Each training "example" = entire dataset
- Input: (X_train, y_train, X_test) → Output: ŷ_test
- Trained ~24 hours on 8 GPUs

Inference

- No gradient updates at test time
- Training data = context tokens
- Test features = query tokens
- Similar to how GPT does few-shot learning

TabPFN: Synthetic Data Prior

Structural Causal Models

DAG-based generative process
Features with causal dependencies
Captures real-world structure

Bayesian Neural Networks

Random NN with weight distributions
Non-linear decision boundaries
Smooth but complex functions

Gaussian Processes

Functions from GP prior
Various kernel functions
Controls target smoothness

By mixing these priors, TabPFN learns to handle diverse data-generating processes.

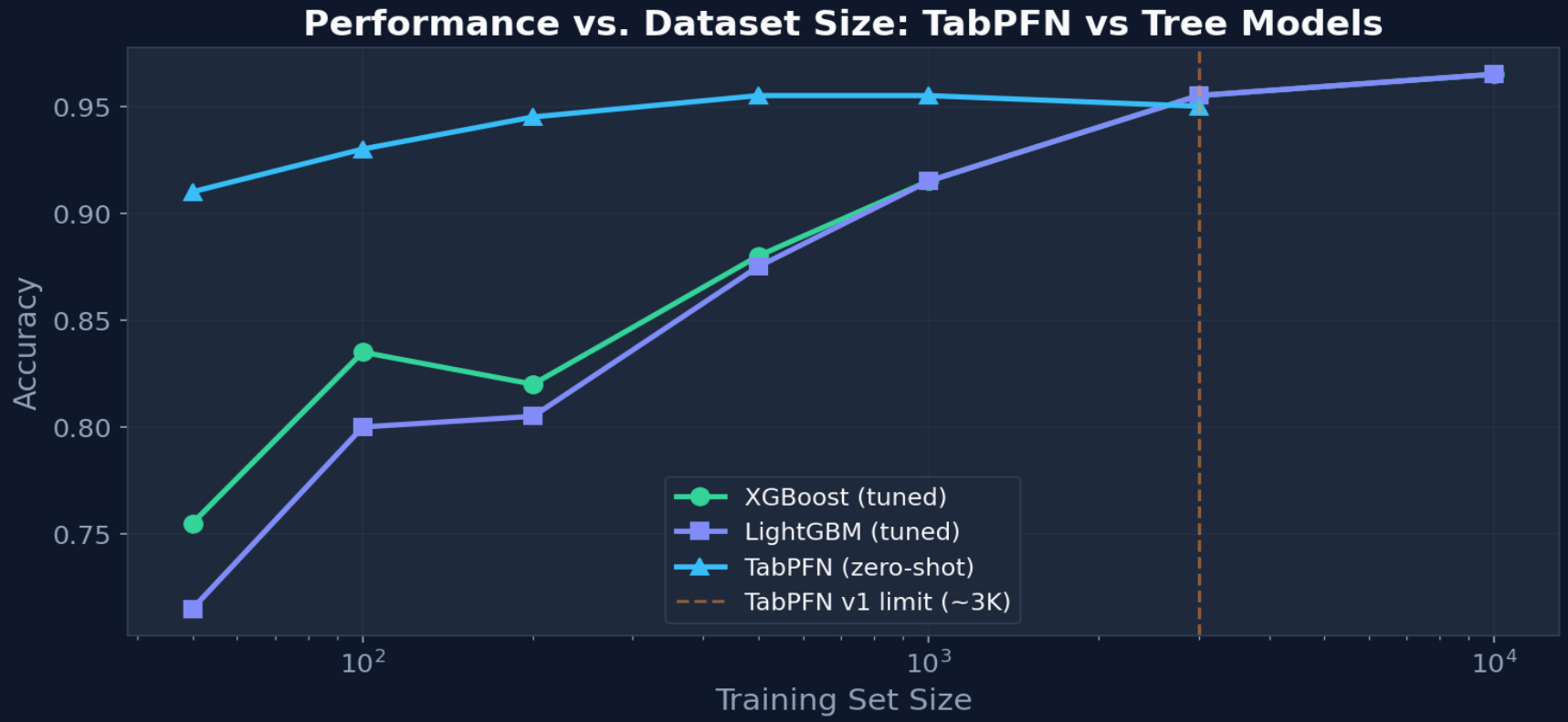
Benchmark: TabPFN vs. Tree-Based Models

Criterion	XGBoost	LightGBM	TabPFN
Small Data (< 1K)	Good	Good	**Best**
Medium (1K–10K)	Strong	Strong	Competitive
Large (> 10K)	**Best**	**Best**	Limited*
HP Tuning	Required	Required	**Not needed**
Training	Sec–min	Sec–min	Single pass
Interpretability	SHAP / FI	SHAP / FI	Limited

** TabPFN v2 (2024) removes many size limits via retrieval-augmented approach + adds regression*

Demo: Performance vs. Dataset Size

TabPFN excels on small data; tree models scale better — the crossover point matters



TabPFN: Code Example

pip install tabpfn — 3 lines of code, zero hyperparameters

```
from tabpfn import TabPFNClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split



# That's it — no hyperparameters to tune!
clf = TabPFNClassifier()
clf.fit(X_tr, y_tr)
preds = clf.predict(X_te)
probs = clf.predict_proba(X_te)  # class probabilities

print(f"Accuracy: {accuracy_score(y_te, preds):.4f}")
# Compare: XGBoost needs tuning, TabPFN just works
```

XGBoost vs. TabPFN: Side-by-Side

XGBoost

```
model = XGBClassifier(
  n_estimators=100,
  max_depth=6,
  learning_rate=0.1,
  subsample=0.8,
  colsample_bytree=0.8,
  reg_alpha=0.1,
  reg_lambda=1.0
)
model.fit(X_tr, y_tr)
# 8 hyperparameters!
```

TabPFN

```
model = TabPFNClassifier()
model.fit(X_tr, y_tr)

# 0 hyperparameters!
```

Same accuracy on small datasets — dramatically less effort

Practical Guide: When to Use What?

Use TabPFN when...

- Small datasets (< 3K samples)
- Quick prototyping / baselines
- No time for HP search
- Classification tasks
- Research / benchmarking

Use XGBoost / LightGBM when...

- Large datasets (> 10K samples)
- Need interpretability (SHAP)
- Production deployment
- Regression tasks at scale
- Established pipelines

Text-to-SQL & Table QA

LLMs can also interact with tabular data through natural language

Text-to-SQL

NL question → LLM → SQL query → DB result

```
Q: "Average salary in engineering?"

SELECT AVG(salary)
FROM employees
WHERE dept = 'engineering';
```

Spider SOTA: ~91% (GPT-4 + agent)

- Prompting: schema + few-shot examples
- Fine-tuned: SQLCoder, NSQL
- Agentic: generate → execute → debug

Table QA

Serialize table → LLM reasons → answer directly

```
Q: "Which product grew most?"

A: Widget A grew 15.6%
   ($45K → $52K, +$7K)
```

No SQL needed — direct reasoning

- TAPAS (Google): BERT for tables
- GPT-4/Claude: prompt-based
- Chain-of-Table: iterative ops

Text-to-SQL: Code Demo with LangChain



LangChain makes it easy to build Text-to-SQL pipelines with any LLM

```
from langchain_community.utilities import SQLDatabase
from langchain_openai import ChatOpenAI
from langchain.chains import create_sql_query_chain

# Connect to database



# Create chain: NL → SQL → execute → answer
chain = create_sql_query_chain(llm, db)

# Ask in natural language
query = chain.invoke({
    'question': 'Top 5 employees by salary in engineering'
})
result = db.run(query)
```

Part 1 Summary: AI for Tabular Data

- Tree-based models (XGBoost, LightGBM) remain the gold standard for most tabular tasks
- TabPFN introduces a foundation model approach: pre-train on synthetic data, predict in one pass
- TabPFN excels on small datasets without tuning; trees dominate on large-scale data
- Text-to-SQL and Table QA enable natural language access to structured data
- The field is moving toward combining structured ML with LLM-based interfaces

PART 2

AI for Time Series



Time Series Data Is Everywhere

Finance

Stock prices, trading volume, economic indicators, risk monitoring

Energy

Power demand forecasting, renewable output, grid optimization

Retail

Sales forecasting, inventory management, demand planning

Healthcare

Patient vitals, disease outbreaks, wearable data

Climate

Weather forecasting, temperature trends, extreme events

IoT / Manufacturing

Sensor data, predictive maintenance, anomaly detection

Time Series Forecasting: Evolution

Classical

~1970s–2000s

ARIMA, ETS,
Exponential Smoothing

ML Era

~2010s

Prophet, XGBoost,
Feature Engineering

Deep Learning

~2018–2023

DeepAR, N-BEATS,
PatchTST, TiDE

Foundation Models

2024–present

TimesFM, Chronos,
Moirai, Moment

Prophet: Decomposition-Based Forecasting

Taylor & Letham, Meta (2017)

Decomposition-Based

Additive: y(t) = trend + seasonality + holidays + noise

Analyst-Friendly

Designed for business analysts — interpretable components

Robust to Messiness

Missing data, outliers, irregular intervals handled gracefully

Domain Knowledge

Easy to add holidays, capacity limits, and changepoints

Prophet: Decomposition Formula

$$y(t) = g(t) + s(t) + h(t) + \varepsilon(t)$$

g(t) — Trend

Piecewise linear or logistic growth
Automatic changepoint detection
Saturating growth option

s(t) — Seasonality

Fourier series approximation
Yearly, weekly, daily patterns
Custom seasonalities

h(t) — Holidays

User-specified holiday effects
Pre/post windows
Country calendars built-in

ε(t) — Residual

Irreducible noise
Laplace or Normal distribution
Uncertainty via MCMC/MAP

Prophet: Basic Usage



```python
from prophet import Prophet
import pandas as pd

# Prophet requires columns: 'ds' (date) and 'y' (value)

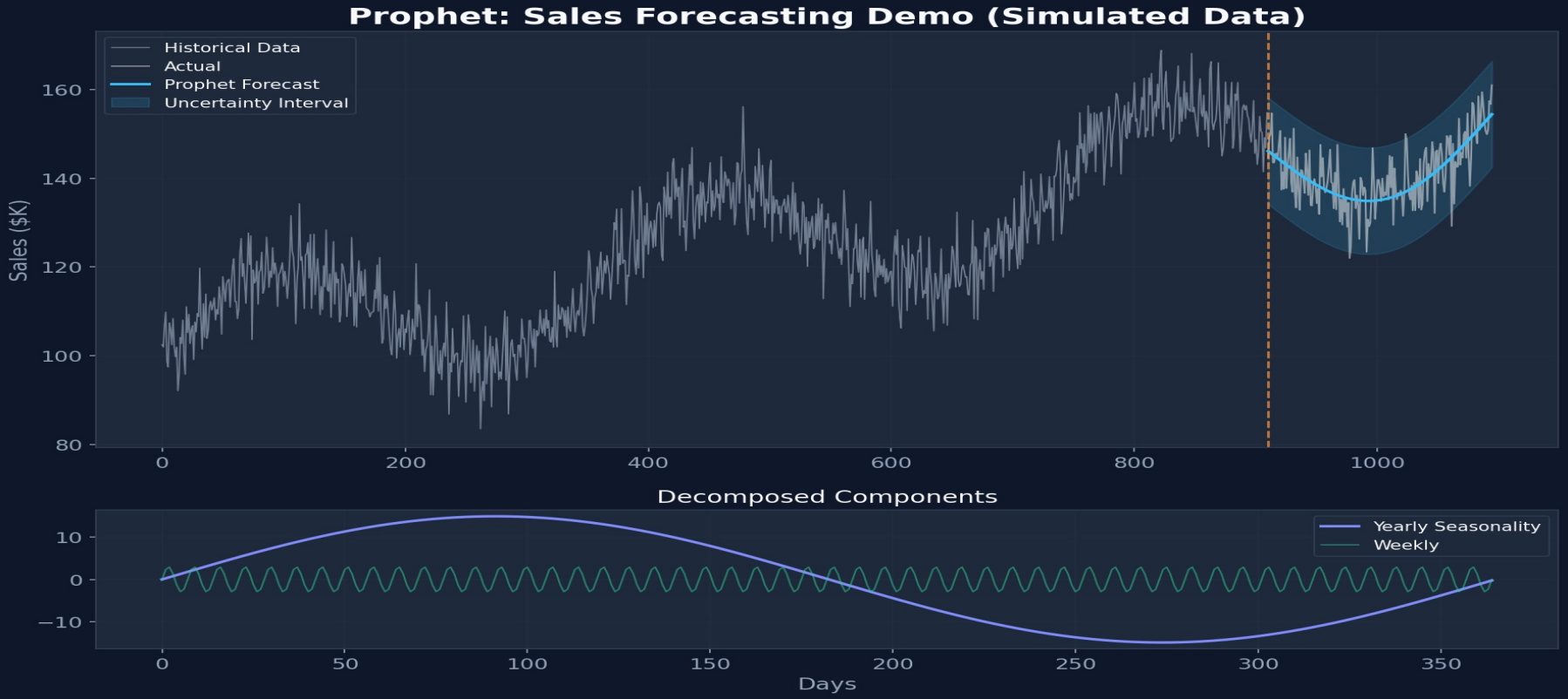


# Fit model
model = Prophet(
    yearly_seasonality=True,
    weekly_seasonality=True,
    changepoint_prior_scale=0.05
)
model.fit(df)

# Forecast next 365 days
future = model.make_future_dataframe(periods=365)
forecast = model.predict(future)
model.plot(forecast)  # built-in viz
```

Demo: Prophet Forecast — Simulated Sales Data

3 years of data → 6-month forecast with uncertainty intervals + decomposed components



Prophet: Strengths & Limitations

Strengths

- Handles missing data & outliers
- Intuitive, interpretable components
- Easy to add domain knowledge
- Works well out-of-the-box
- Scales to many series (batch)
- Uncertainty intervals built-in

Limitations

- Univariate only (no multivariate)
- No cross-series learning
- Struggles with complex patterns
- Doesn't leverage covariates well
- No transfer learning across domains
- Can overfit changepoints

NeuralProphet (2021) addresses some of these limits — PyTorch backend, AR-Net, covariate support — same API feel.

Foundation Models for Time Series: Why Now?

Traditional

Train one model per dataset
No knowledge transfer
Requires domain-specific tuning

Foundation Model

Pre-train on massive TS corpus
Zero-shot to new domains
Cross-domain generalization

Why now?

Data availability

Large public TS datasets (Google Trends, weather, finance, Wiki)

Transformer architecture

Proven ability to model sequences across modalities

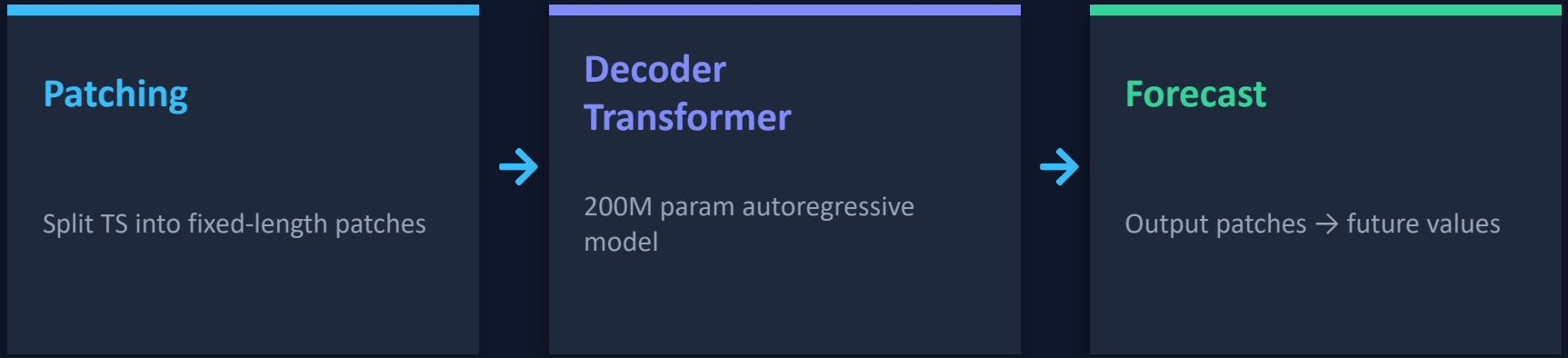
Scaling laws

Bigger models + more data = better zero-shot performance

Foundation Models: Landscape

Model	Org	Architecture	Key Idea
TimesFM	Google	Decoder Transformer	Patching + pre-train on 100B+ points
Chronos	Amazon	T5 (Enc-Dec)	Tokenize values → language modeling
Moirai	Salesforce	Masked Enc	Any-variate, any-frequency, unified
Moment	CMU	Masked Enc	Multi-task pre-training on patches
Lag-Llama	ServiceNow	LLaMA-style	Probabilistic forecasting with lags

TimesFM (Google, 2024)



- Pre-trained on 100B+ time points: Google Trends, Wiki, synthetic data
- Zero-shot beats many supervised baselines on Monash benchmark
- Variable context length and forecast horizons

TimesFM: Code Example

pip install timesfm — zero-shot forecasting from a pre-trained checkpoint

```
import timesfm
import numpy as np

# Load pre-trained model from HuggingFace
tfm = timesfm.TimesFm(
    context_len=512,
    horizon_len=128,
    input_patch_len=32,
    output_patch_len=128,
    num_layers=20,
    model_dims=1280,
)
tfm.load_from_checkpoint(
    repo_id='google/timesfm-1.0-200m')

# Zero-shot forecast — no training needed!
forecast = tfm.forecast(my_time_series_array)
```

Chronos (Amazon, 2024)

Core Idea: Treat forecasting as language modeling — tokenize values into discrete bins, predict next tokens.

1 Scaling

Normalize by mean abs value

2 Tokenize

Quantize into 4096 bins

3 T5 Model

Cross-entropy on token seq

4 Sample

Multiple paths → uncertainty

Model sizes: Mini (20M), Small (46M), Base (200M), Large (710M) — all based on T5

Chronos: Code Example



pip install chronos-forecasting — probabilistic forecasting with uncertainty

```
import torch
from chronos import ChronosPipeline
import pandas as pd

# Load pre-trained Chronos
pipeline = ChronosPipeline.from_pretrained(
    'amazon/chronos-t5-base',
    device_map='auto',
    torch_dtype=torch.bfloat16
)

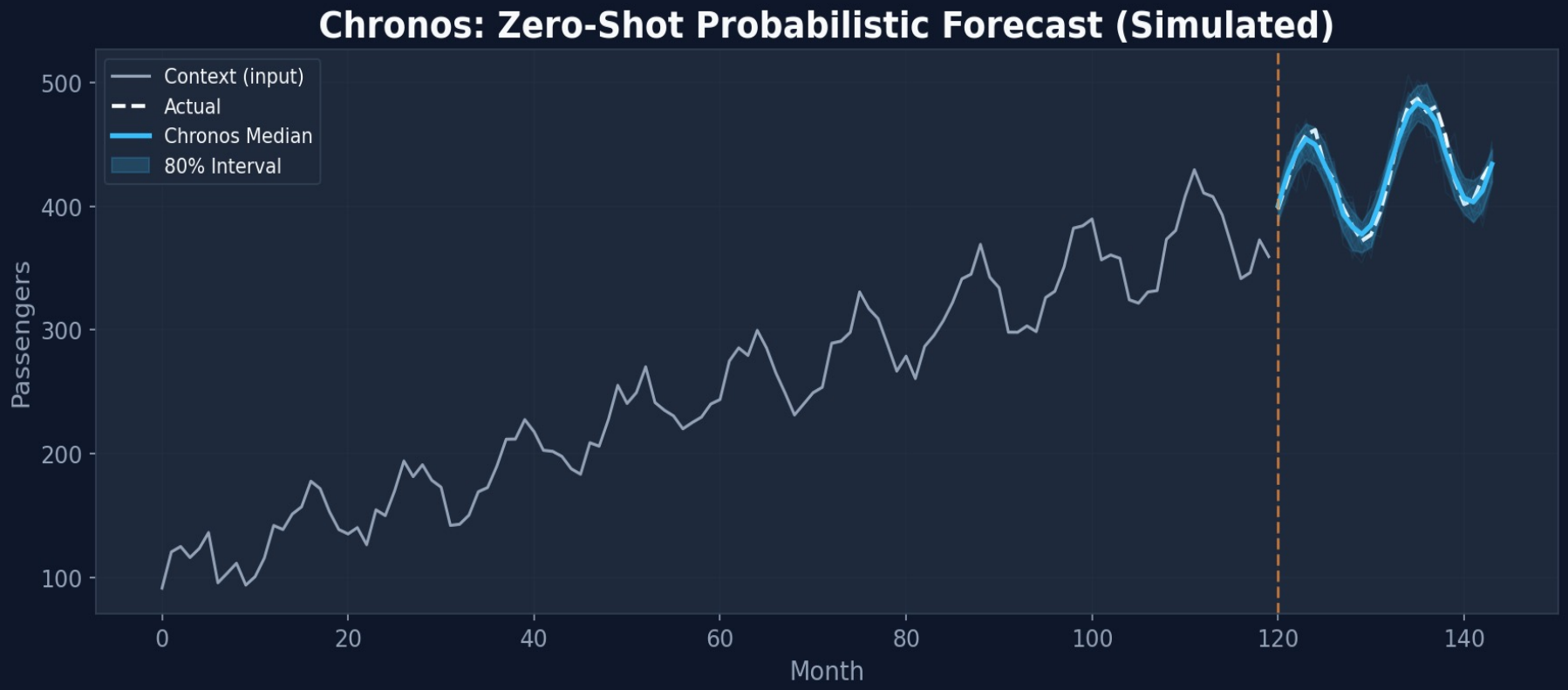
# Load data and forecast



# Probabilistic forecast — returns multiple samples!
forecast = pipeline.predict(context, prediction_length=30,
                            num_samples=100)  # 100 paths
```

Demo: Chronos — Probabilistic Zero-Shot Forecast

100 sampled forecast paths → median + 80% prediction interval — no training on this data!



Moirai (Salesforce, 2024)

Any-Variate

Univariate and multivariate in one model

Any-Frequency

Minutely to yearly via multi-patch projection

Flexible Horizons

Arbitrary prediction lengths without re-training

LOTSA Data

27B observations from 9 diverse domains

Comparing Time Series Foundation Models

	TimesFM	Chronos	Moirai
Architecture	Decoder-only	T5 (Enc-Dec)	Masked Encoder
Tokenization	Patch embedding	Scalar quantization	Patch embedding
Multivariate	Channel-indep.	Univariate only	**Native**
Probabilistic	Point forecasts	**Yes (sampling)**	**Yes (mixture)**
Pre-training Data	100B+ points	Public TS + augm.	27B (LOTSA)
Open Source	Yes	Yes	Yes

Practical Guide: Choosing a TS Approach

Scenario	Recommended	Why
Quick baseline	**Prophet**	Easy, interpretable, robust
Need covariates	**Prophet + features**	Add regressors to Prophet or use XGBoost
Zero-shot, new domain	**TimesFM / Chronos**	Pre-trained, no target data needed
Uncertainty needed	**Chronos**	Native probabilistic via sampling
Multivariate	**Moirai**	Any-variate natively
Production + interpretable	**Prophet + XGBoost**	Proven, mature, well-supported

Start simple (Prophet), validate with FMs (TimesFM/Chronos), decide based on your needs.

Prophet vs. Chronos: Side-by-Side

Prophet

```
model = Prophet(
  yearly_seasonality=True
)
model.fit(df)  # needs training

future = model.make_future_
  dataframe(periods=30)
forecast = model.predict(
  future)
# Point forecasts + intervals
```

Chronos

```
pipe = ChronosPipeline
  .from_pretrained(
    'amazon/chronos-t5-base'
  )
# No training needed!
forecast = pipe.predict(
  context,
  prediction_length=30,
  num_samples=100)
# 100 probabilistic paths
```

Prophet: train per dataset, interpretable | Chronos: zero-shot, probabilistic

Part 2 Summary: AI for Time Series

- Prophet: interpretable, robust baseline — still widely used in industry
- Foundation models (TimesFM, Chronos, Moirai) enable zero-shot forecasting
- Chronos tokenizes TS values — treating forecasting as language modeling
- Moirai handles any-variate, any-frequency with a single unified model
- Start simple (Prophet), validate with FMs, then decide based on your constraints

Key Takeaways

- Foundation models are expanding to every data modality — tabular and time series are the latest frontiers
- TabPFN: meta-learning matches XGBoost on small datasets without any tuning
- Prophet remains a strong, interpretable baseline — know when simple models are enough
- TimesFM / Chronos bring "pre-train once, forecast anything" to time series
- The best approach depends on data size, task, and production constraints — no universal winner