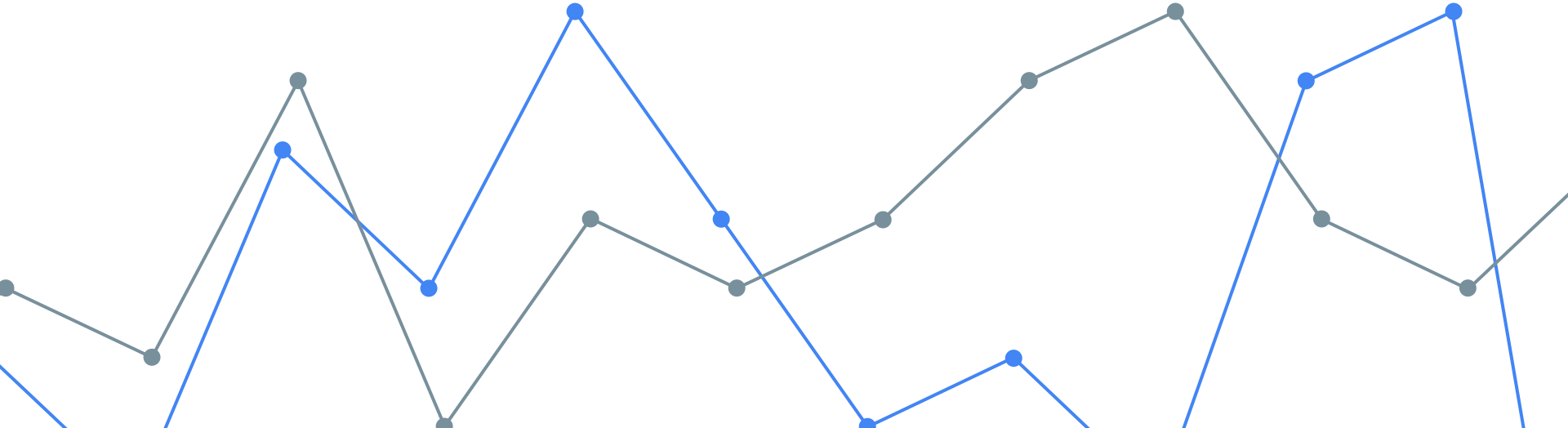
Neural Networks vs. Classical Methods

By Charles Dilger & Ahsan Imran



Dataset | NY Housing

Our first dataset will be on housing in NY.

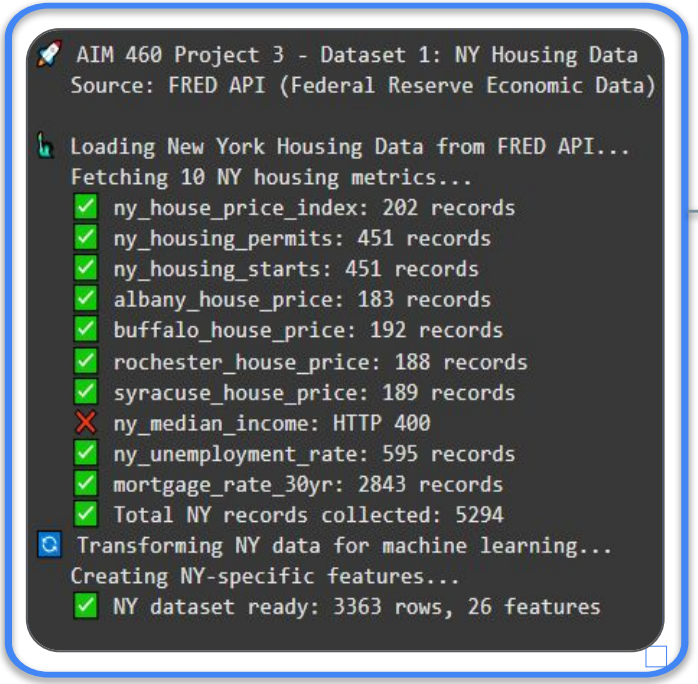
It comes from the Federal Reserve Economic Data (FRED) API.

Our target variable will be average housing price.

It will be an engineered feature, taking the mean of all cities' housing prices.

This dataset contains 12 rows that are missing values.

It contains features such as the date, cities' housing prices, mortgage rate, NY housing permits, NY price growth, and more.



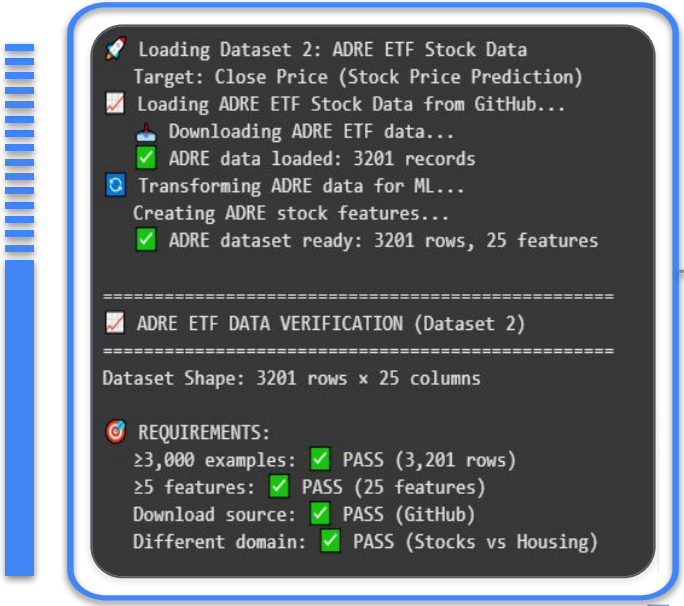
Dataset | ADRE ETF

Our second database is the ADRE ETF, available at Kaggle's Huge Stock Market Dataset.

It is the value of BLDRS Emerging Markets 50 ADR Index Fund.

It contains the opening value, closing value, high, low, volume, and return.

We will try to predict the closing value, based on the other features.



NY Housing EDA Pt.1

Result 1

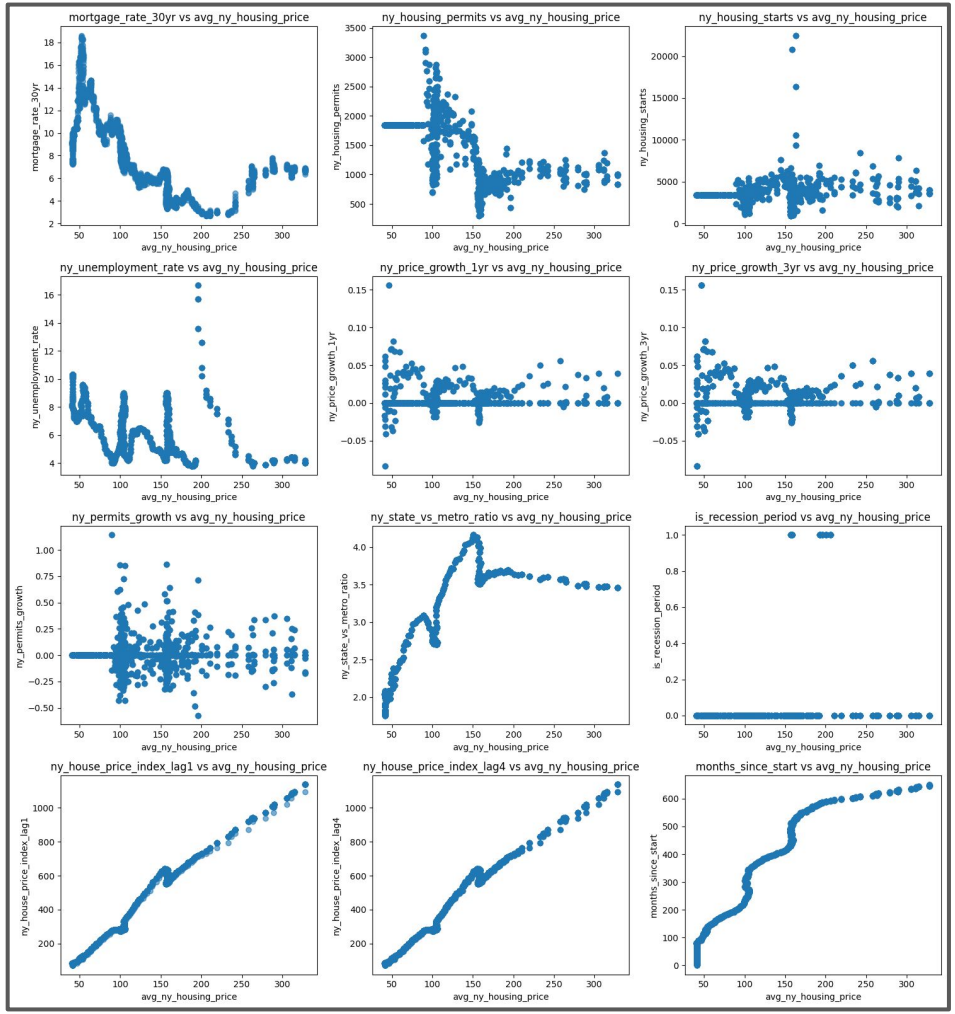
Engineered a target feature called Average NY Housing Price, that takes the mean of all cities' housing prices.

Result 2

Engineered a feature called Months Since Start, which counts the months past since the beginning of the data.

Result 3

12 rows were removed for having missing values, leaving 3350 rows



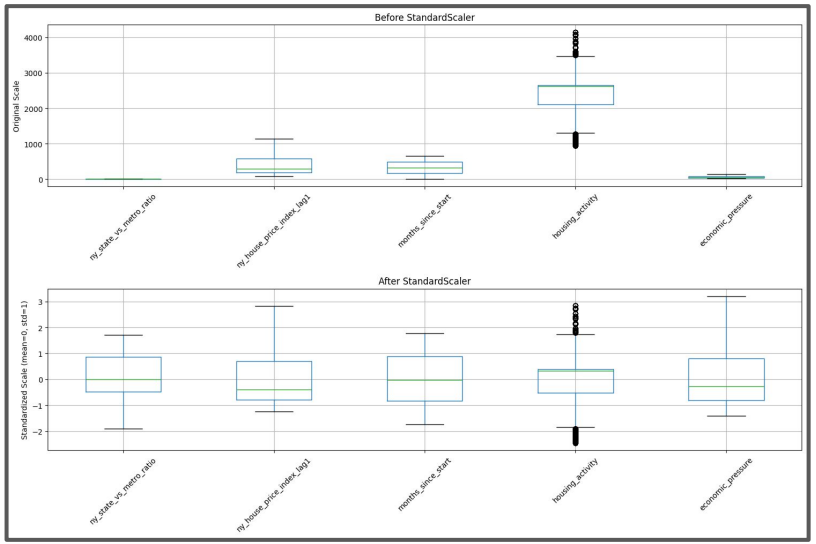
NY Housing EDA Pt.2

From then on, we:

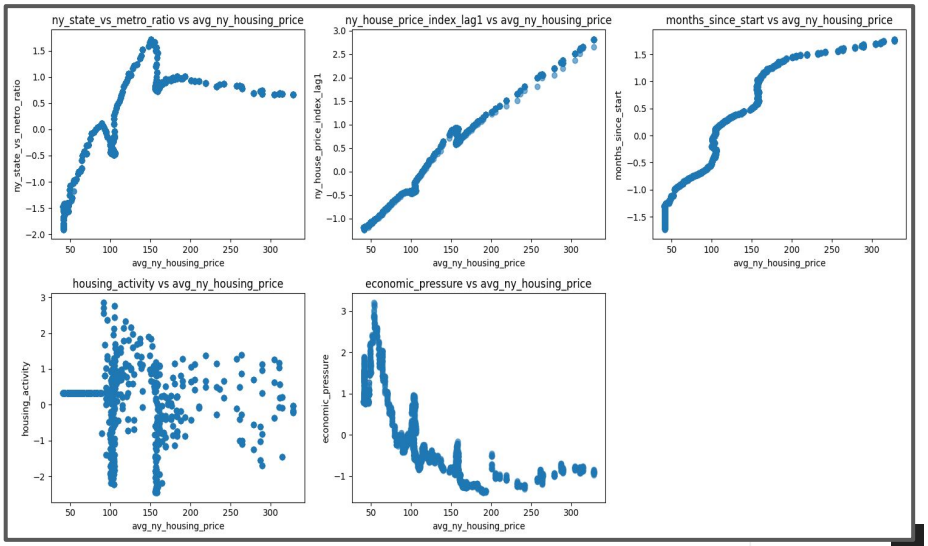
- Removed seemingly redundant features.
- Applied IQR in order to remove outliers.
- Checked the VIF score, found a score as high as 81.
- Decided to engineer 2 new features:
  - Housing Activity, the combination of NY Housing Permits and NY Housing Starts
  - Economic Pressure, the combination of NY Unemployment Rate and 30-Year Mortgage Rate

```
Combining highly correlated features to reduce multicollinearity...
Created: housing_activity (average of permits and starts)
Created: economic_pressure (unemployment x mortgage rate)

Removing original features: ['ny_housing_permits', 'ny_housing_starts', 'ny_unemployment_rate', 'mortgage_rate_30yr']

Final feature set (5 features):
 1. ny_state_vs_metro_ratio
 2. ny_house_price_index_lag1
 3. months_since_start
 4. housing_activity
 5. economic_pressure
```

NY Housing StandardScaler

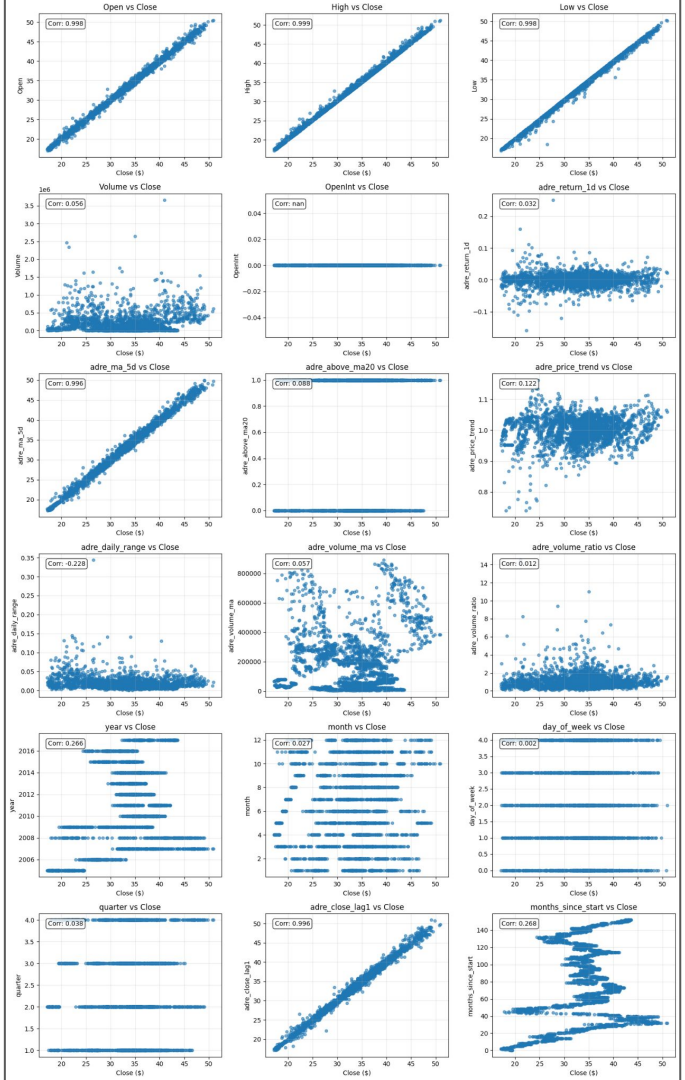


Pre-scaling Scatterplot



ADRE ETF Dataset Pt.1

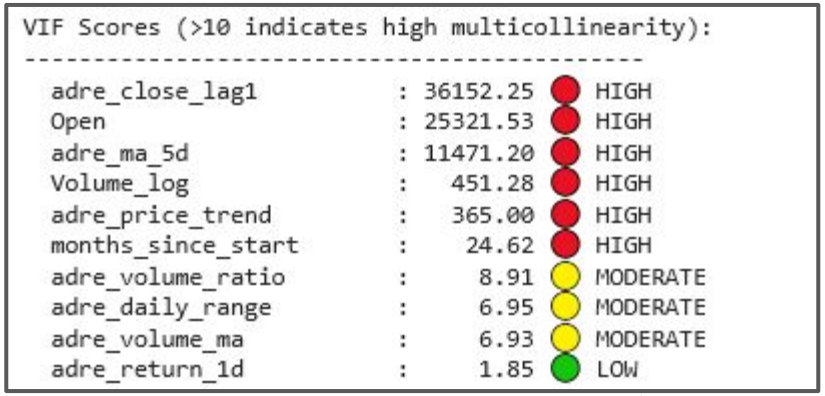
No missing values were found.
We removed a few redundant features right away, such as delayed returns and lagging variables, that would likely have extreme multicollinearity.
After plotting, we removed weak correlations and time bands.



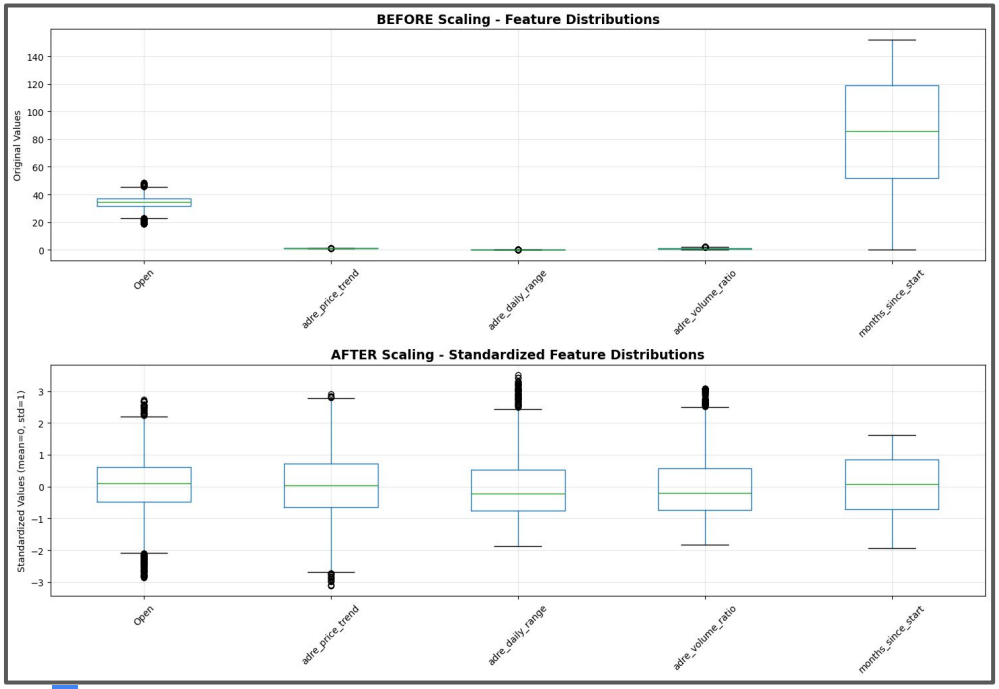
ADRE ETF Dataset Pt.2

Volume had a skew, so we applied a logarithmic transformation to it.
Our VIF scores were insanely high, so we removed adre_close_lag1 and adre_ma_5d. We left Open because we wanted to predict the closing stock value based primarily on the opening stock value. It is the most important part of our models.

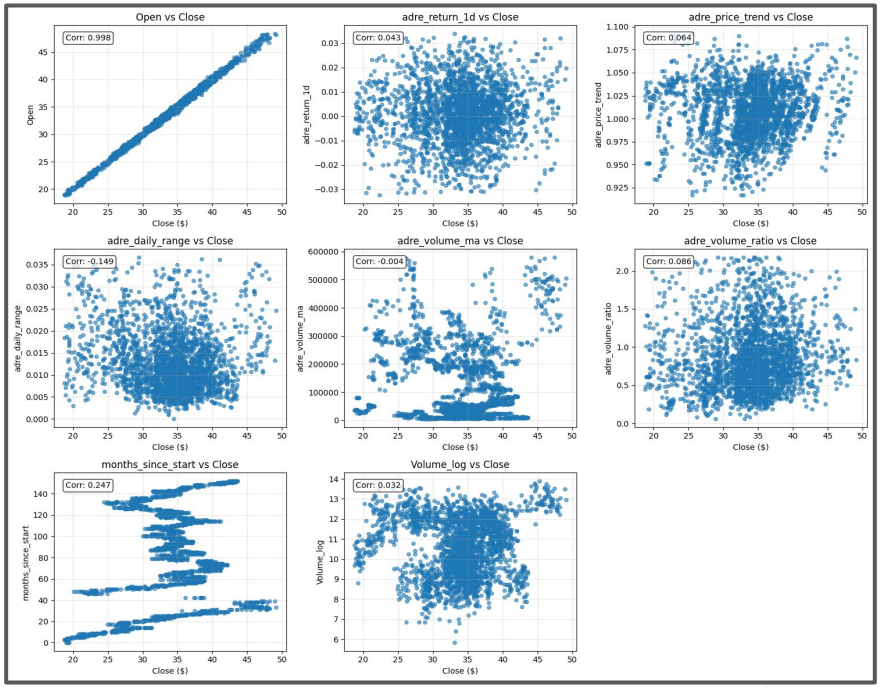
We then removed poorly correlated features.



ADRE StandardScaler

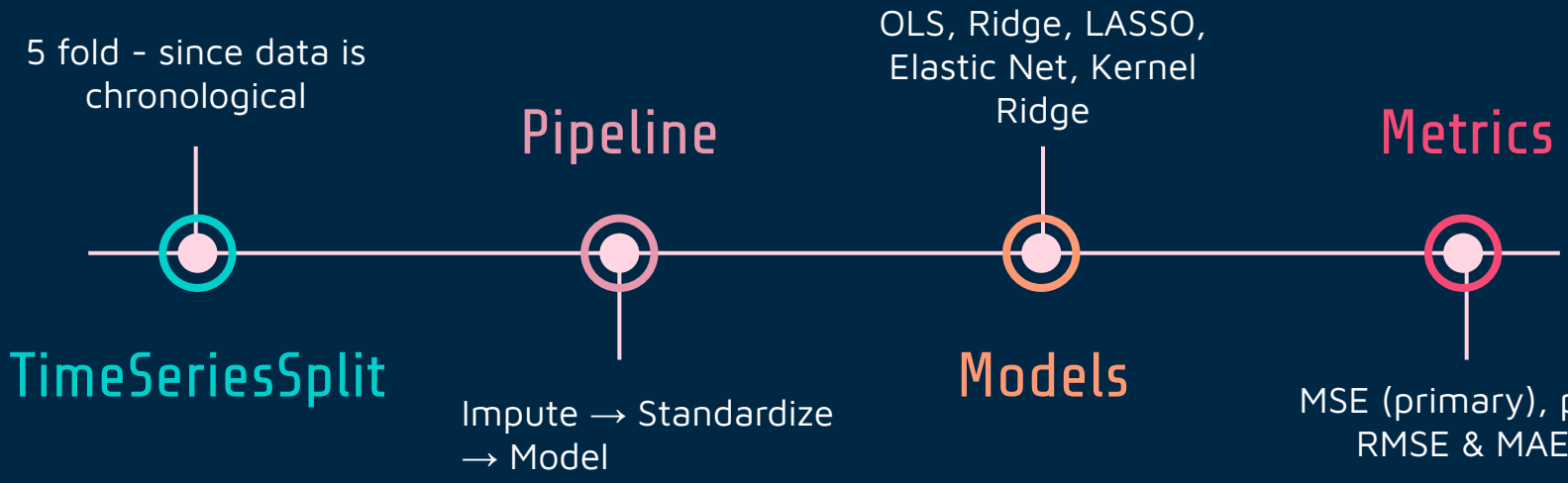


Pre-scaling Scatterplot



Baseline Regression & Kernel Methods

Evaluation Setup



Linear Baseline

Model	What it does	How we tuned	When it helps
OLS	Reference fit & No Penalty	-	Quick floor to beat
Ridge	Shrinks all coeffs	$\alpha \in \{0.01, 0.1, 1, 10, 100\}$	Features correlated
LASSO	Drives Coeff to 0	$\alpha \in \{0.0005, 0.001, 0.01, 0.1, 1\}$	feature selection via sparsity
Elastic Net	Balance of shrink & section	$\alpha \in \{0.001, 0.01, 0.1, 1\}$, l1_ratio $\in \{0.1 \ldots 0.9\}$	Correlated features + some pruning

Linear Baselines Results

```
Using TimeSeriesSplit for housing data
Top 10 Linear Models:
    family       model     val_MSE   val_RMSE    val_MAE  fit_time_s   alpha
15  Linear  ElasticNet  487.505795  16.342489  12.919377    0.011601  0.0010
1   Linear       Ridge  499.175165  17.124968  13.385103    0.004056  0.0100
2   Linear       Ridge  500.678205  17.352979  13.761011    0.007477  0.1000
7   Linear       LASSO  507.594547  17.602253  13.778482    0.010070  0.0010
8   Linear       LASSO  512.510929  17.672512  14.039539    0.011123  0.0100
6   Linear       LASSO  512.975345  17.827460  13.955985    0.007461  0.0005
0   Linear         OLS  518.903167  18.053405  14.135162    0.006170     NaN
14  Linear  ElasticNet  515.261795  18.128285  14.495841    0.010756  0.0010
13  Linear  ElasticNet  534.980997  18.840802  15.147588    0.014883  0.0010
12  Linear  ElasticNet  548.841016  19.236168  15.527127    0.012880  0.0010
```

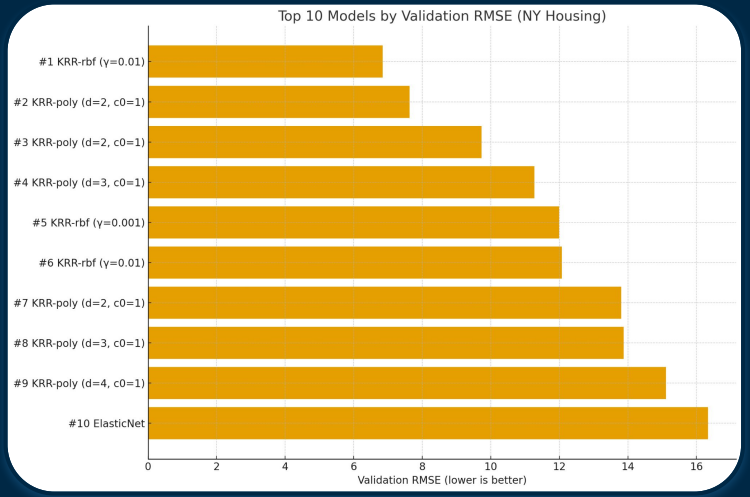
Kernel Ridge

Since our data isn't Linear, Our dataset is benefiting from the kernel methods in comparison to our linear models

Our Top 5 Kernel Models:

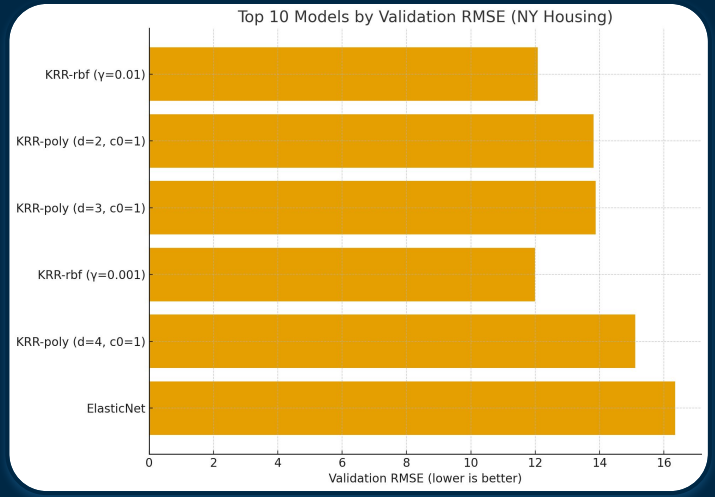
	family	model	alpha	gamma	val_MSE	val_RMSE	val_MAE
1	Kernel	KRR-rbf	0.001	0.010	88.991282	6.842321	5.061987
26	Kernel	KRR-poly	0.001	NaN	65.024493	7.628858	5.467877
32	Kernel	KRR-poly	0.010	NaN	135.964599	9.734397	6.869843
28	Kernel	KRR-poly	0.001	NaN	174.701644	11.271754	8.704205
0	Kernel	KRR-rbf	0.001	0.001	182.589075	11.993725	9.898700

Top 10 Models by Validation RMSE



KRR-RBF

Kernel Ridge with an RBF kernel clearly wins on validation with RMSE



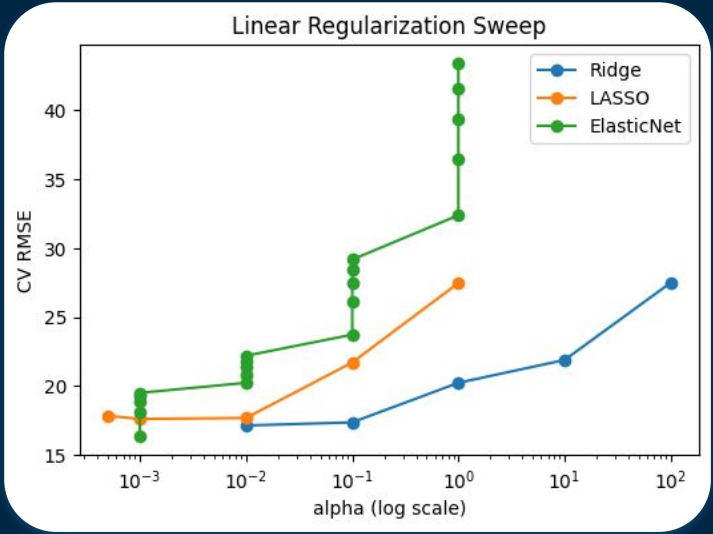
What the data shows

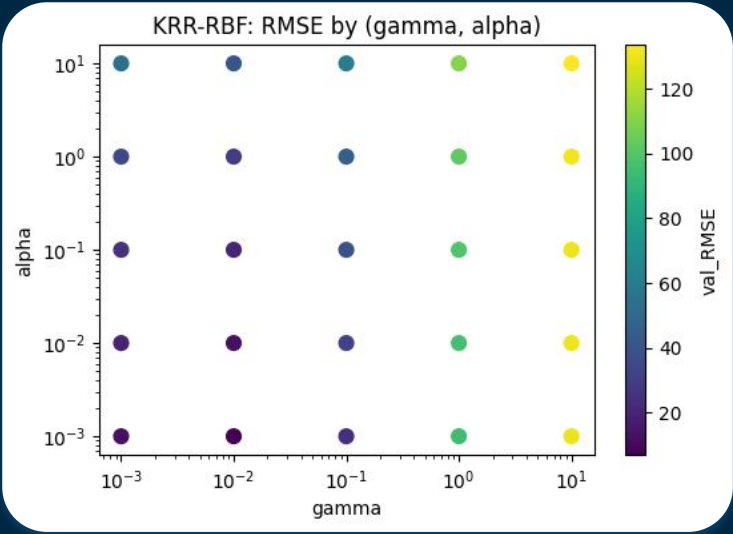
Our recommended default RBF KRR with a quick γ sweep around 0.001-0.01

Regularization Sweep

&

KRR–RBF Landscape





Feedforward Neural Networks

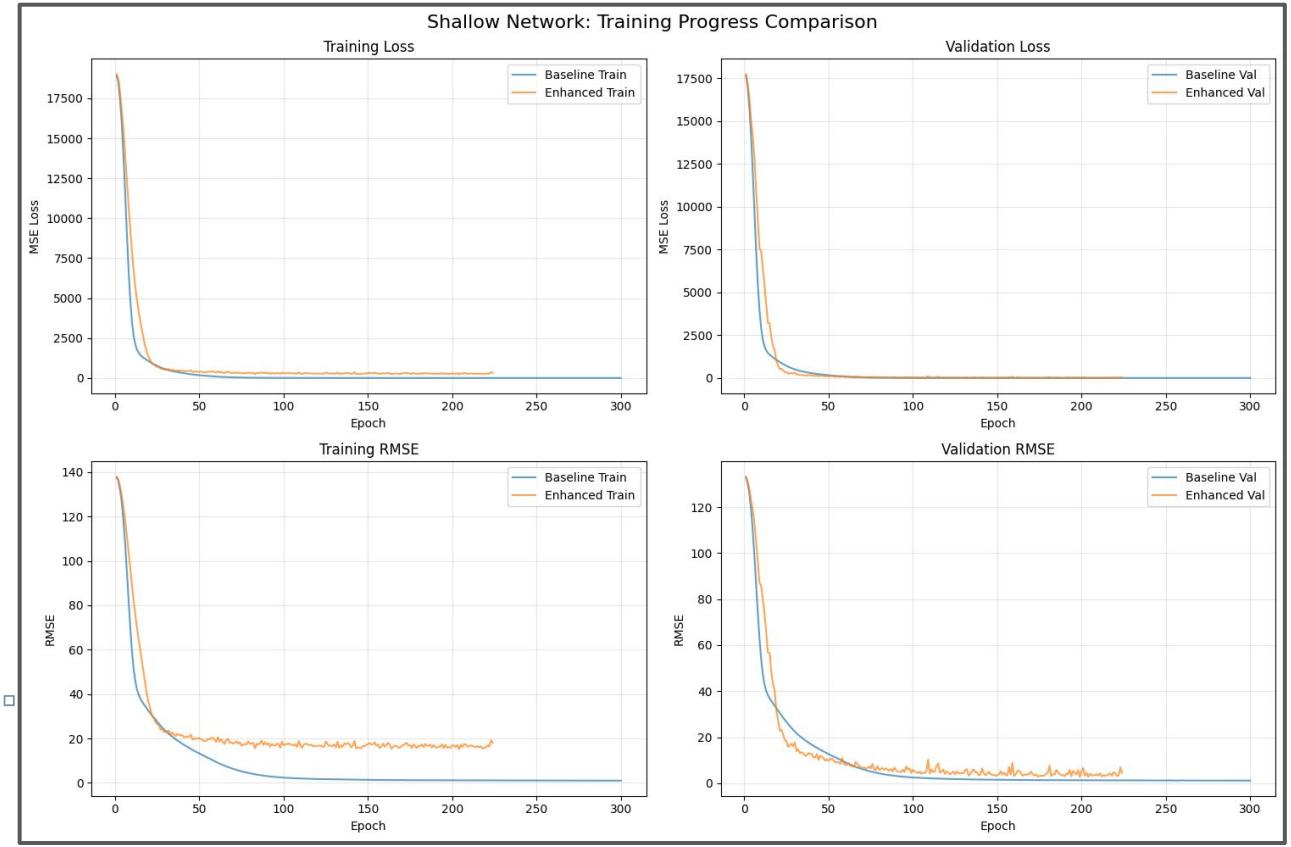
Shallow Neural Network

Our first neural network was a shallow neural network.
We did a baseline neural network of 64 neurons, and then we did an enhanced version with batch normalization, a 20% dropout rate, and replacing ReLU with Swish.

The baseline outperformed, with a RMSE of 0.9466 compared to the enhanced's 4.5857.

```
=== Shallow Network Architecture ===
Baseline Shallow Net:
ShallowNet(
  (hidden): Linear(in_features=5, out_features=64, bias=True)
  (output): Linear(in_features=64, out_features=1, bias=True)
)
Parameters: 449

Enhanced Shallow Net:
ShallowNet(
  (hidden): Linear(in_features=5, out_features=64, bias=True)
  (output): Linear(in_features=64, out_features=1, bias=True)
  (batch_norm): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
Parameters: 577

Forward pass test:
Input shape: torch.Size([32, 5])
Baseline output shape: torch.Size([32, 1])
Enhanced output shape: torch.Size([32, 1])
✅ Architecture working correctly!
```

Shallow Neural Network - Results



Deep Neural Network

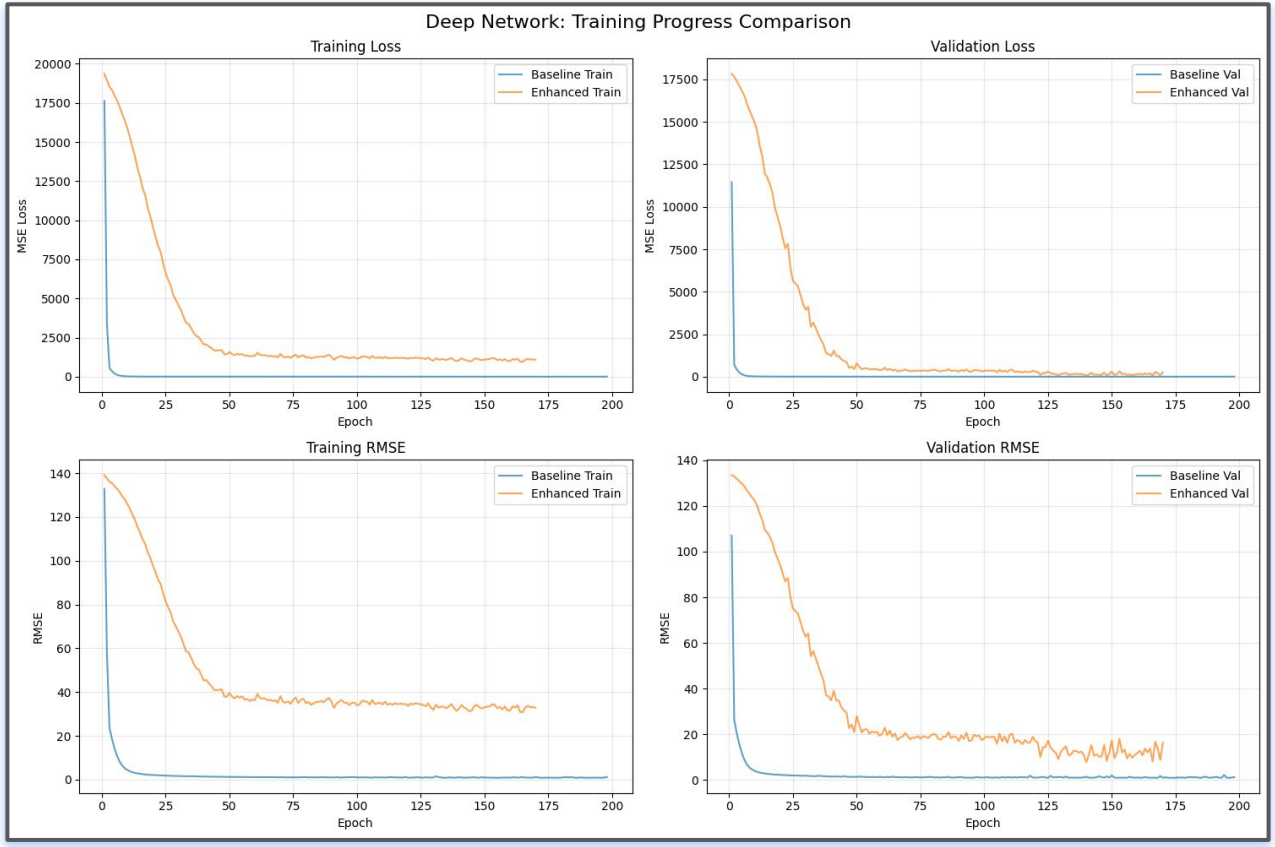
We repeated the process with a deep neural network. This model had 4 layers; they were 128, 64, 32, and 16 neurons, respectively. The baseline got a RMSE of 1.0305, worse than the shallow neural network.
We tried an enhanced version, with Mish activation, batch normal layers, and 30% dropout.

This performed significantly worse due to over-regularization. It had a RMSE of 16.8254.

```
=== Deep Network Architecture ===
Baseline Deep Net:
DeepNet(
  (hidden_layers): ModuleList(
    (0): Linear(in_features=5, out_features=128, bias=True)
    (1): Linear(in_features=128, out_features=64, bias=True)
    (2): Linear(in_features=64, out_features=32, bias=True)
    (3): Linear(in_features=32, out_features=16, bias=True)
  )
  (batch_norm_layers): ModuleList()
  (output): Linear(in_features=16, out_features=1, bias=True)
)
Parameters: 11,649
```

```
Architecture Summary:
Hidden layers: 4 (meets ≥4 requirement)
Layer sizes: 5 → 128 → 64 → 32 → 16 → 1
✅ Deep architecture working correctly!
```

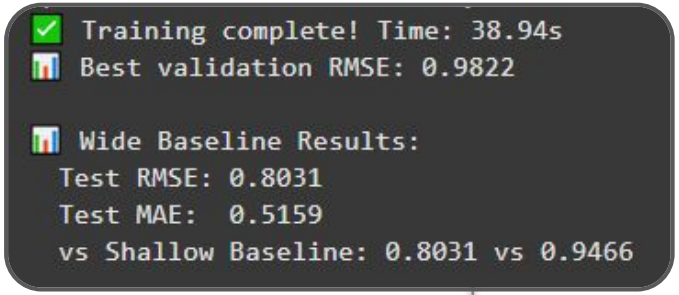
Deep Neural Network - Results

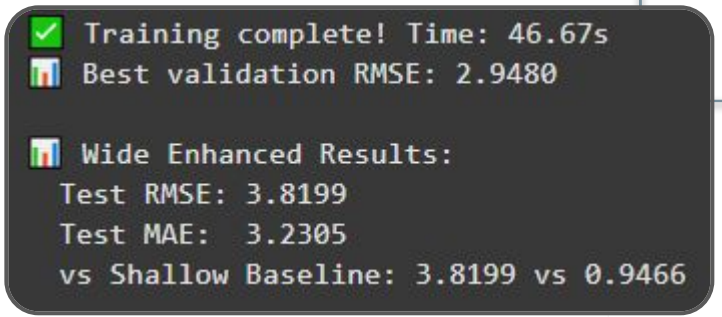


Wide Neural Network

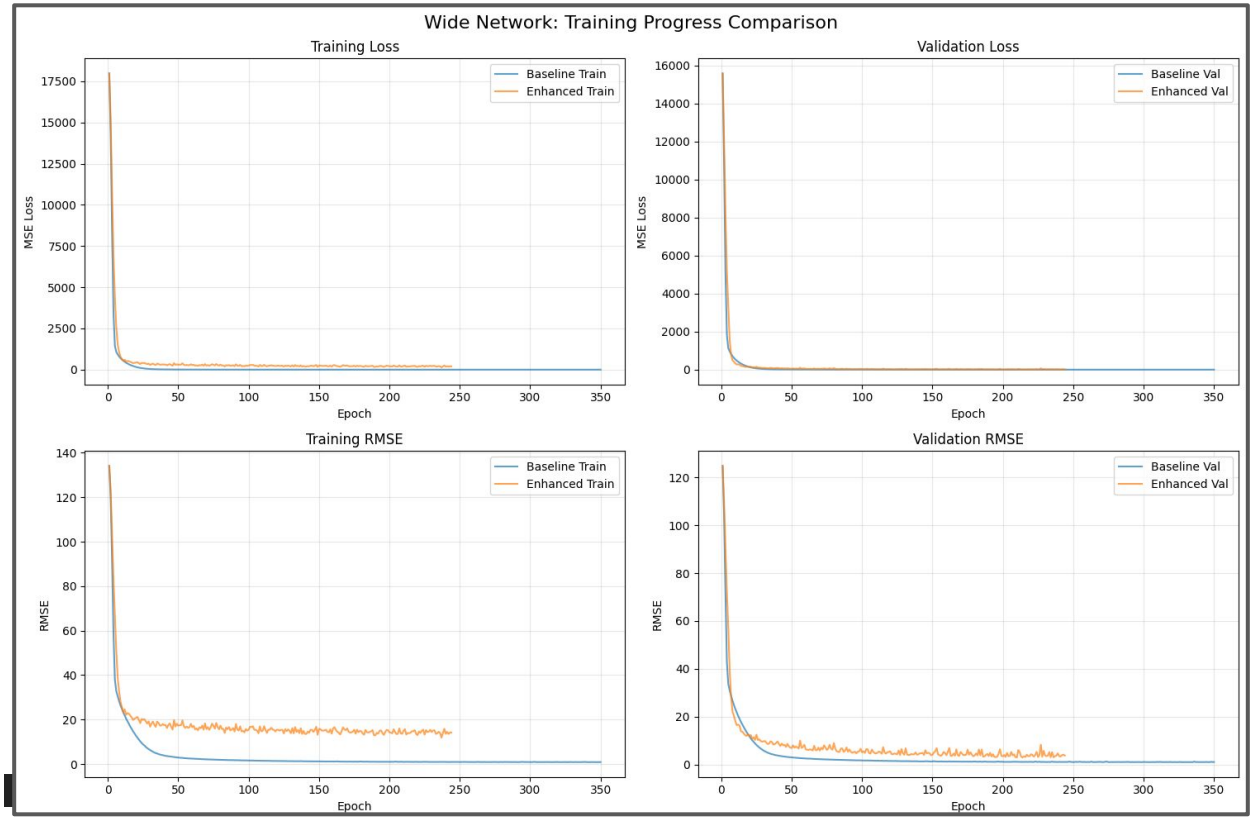
This time, we tried it with a model that had 1 hidden layer of 512 neurons.
This performed shockingly well, with a RMSE of only 0.8031, our lowest yet.

We tried an enhanced version, as well. It had Swish activation, batch norm, 40% dropout.
As expected, this performed worse, with a RMSE of 3.8199. It was still the best performer of all enhanced models, further asserting that a wide neural network is the best fit for our dataset.





Wide Neural Network Results



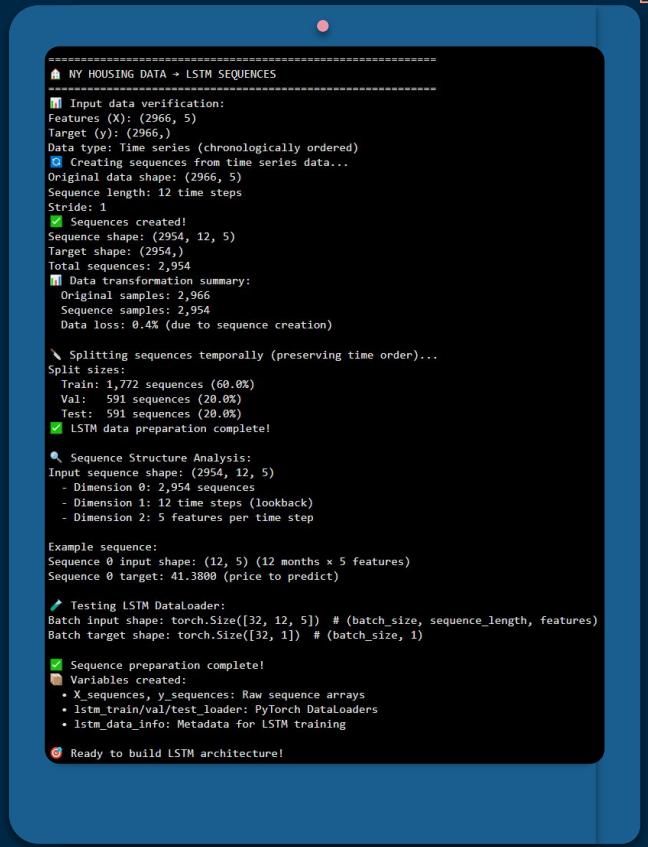
Recurrent Neural Networks

Recurrent Neural Networks

For our RNN, we will be using our NY Housing data. It is chronologically ordered, had explicit time progression, has lag features, momentum, and cycles. It is an ideal dataset for running an RNN.

We will be using LSTM instead of GRU, due to long-term memory needs, and the complex relationships. LSTM is better at learning long-term, complex patterns.

We first implemented Sliding Window Transformation to convert our chronological housing data into sequences.

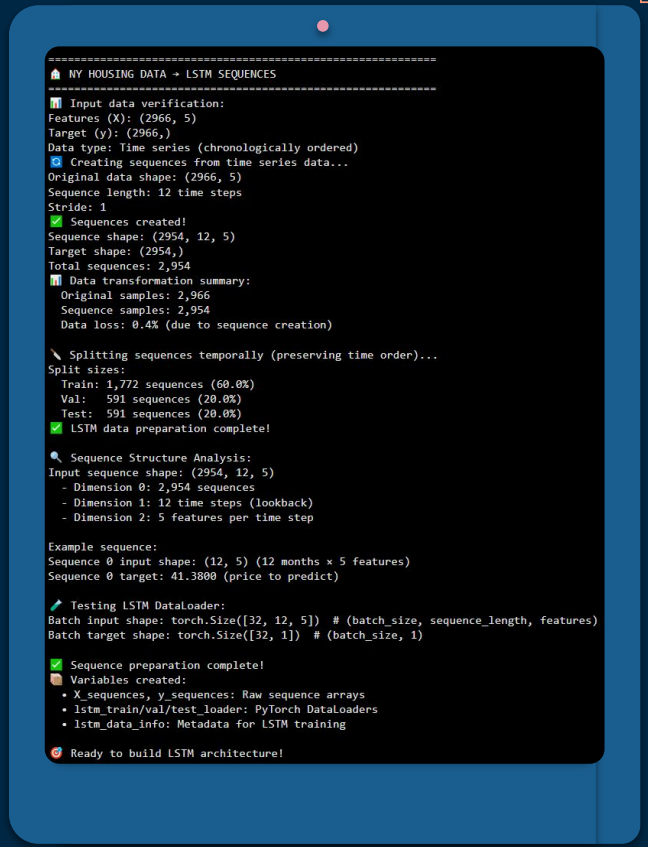


Recurrent Neural Networks

For our RNN, we will be using our NY Housing data. It is chronologically ordered, had explicit time progression, has lag features, momentum, and cycles. It is an ideal dataset for running an RNN.

We will be using LSTM instead of GRU, due to long-term memory needs, and the complex relationships. LSTM is better at learning long-term, complex patterns.

We first implemented Sliding Window Transformation to convert our chronological housing data into sequences.



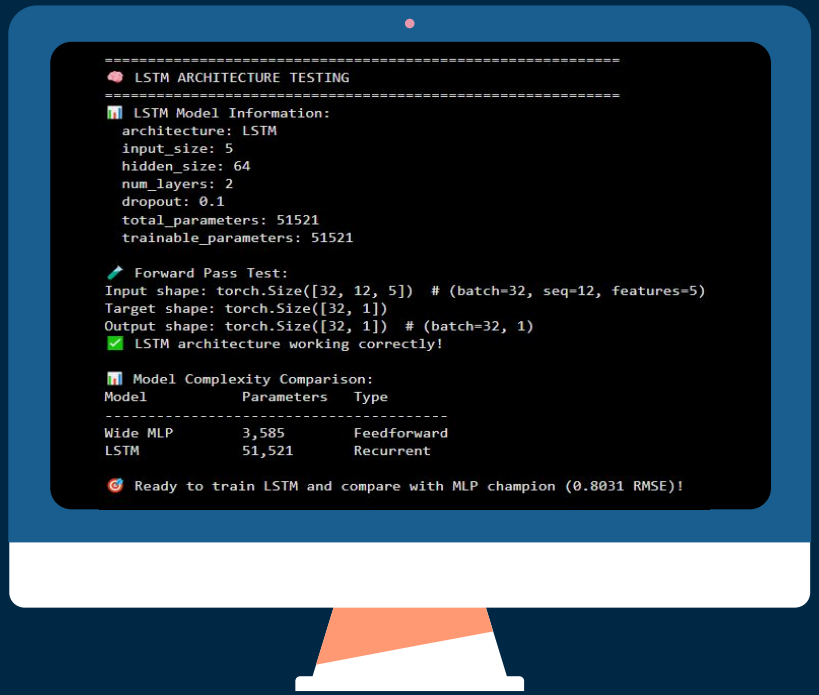
Recurrent Neural Networks Pt.2

We ended up with a shocking 51,521 parameters to work with in our RNN model.

This added complexity backfired, as our model was gravely overfitted.

We ended up with a RMSE of 5.0388, much higher than our greatest performing neural network.

It even took much longer to run the model.



Feature-Transfer Experiment

Feature-Transfer Experiment

We checked whether or not the learned representations of our best performer, the wide baseline network, can outperform our original features. We tested this on both the same domain, our housing dataset, and a different domain, our ADRE stock.

Our model had remarkable ability to generalize. The neural features we extracted from the housing-trained model improved the prediction power of our stock by 16.1-37.3% when implemented on our ADRE model. The model performed excellently on ADRE, maintaining a RMSE of 0.1889 on the Kernel+Neutral learning test.

Feature-Transfer Experiment output

```
============================================================
🎯 FEATURE-TRANSFER LEARNING EXPERIMENT
============================================================
🏆 Using Wide Baseline Network (0.8031 RMSE champion)
🔄 Extracting 512-dimensional learned representations
📊 Comparing transfer performance vs original features

🏠 Transfer Learning Experiment 1: NY Housing (same dataset)

============================================================
🔄 TRANSFER LEARNING: NY HOUSING
============================================================
🔍 Extracting features from hidden layer...
✅ Extracted features shape: (2966, 512)
🔍 Extracting features from hidden layer...
✅ Extracted features shape: (2966, 512)
📊 Feature extraction summary:
  Source features: (2966, 512)
  Target features: (2966, 512)
  Original → Neural features: 5 → 512

🔍 Baseline: Direct regression on original NY Housing features
  Baseline RMSE: 22.3422 ± 26.3850

🔄 Transfer Learning: Ridge on neural network features
  Transfer RMSE: 5.9297 ± 5.8843

🧠 Advanced Transfer: RBF Kernel on neural features
  Kernel Transfer RMSE: 99.2762 ± 115.3576

📊 NY Housing Transfer Learning Results:
Method               RMSE       Improvement
--------------------------------------------
Original Features    22.3422    —
Neural Features      5.9297     +73.5%
Kernel+Neural        99.2762    -344.3%

📈 Transfer Learning Experiment 2: ADRE Stock (cross-domain)
📊 ADRE dataset: 2563 samples, 5 features
```

```
============================================================
🔄 TRANSFER LEARNING: ADRE STOCK
============================================================
🔍 Extracting features from hidden layer...
✅ Extracted features shape: (2966, 512)
🔍 Extracting features from hidden layer...
✅ Extracted features shape: (2563, 512)
📊 Feature extraction summary:
  Source features: (2966, 512)
  Target features: (2563, 512)
  Original → Neural features: 5 → 512

🔍 Baseline: Direct regression on original ADRE Stock features
  Baseline RMSE: 0.3013 ± 0.0000

🔄 Transfer Learning: Ridge on neural network features
  Transfer RMSE: 0.2527 ± 0.0000

🧠 Advanced Transfer: RBF Kernel on neural features
  Kernel Transfer RMSE: 0.1889 ± 0.0000

📊 ADRE Stock Transfer Learning Results:
Method               RMSE       Improvement
--------------------------------------------
Original Features    0.3013     —
Neural Features      0.2527     +16.1%
Kernel+Neural        0.1889     +37.3%

======================================================================
🎓 TRANSFER LEARNING CONCLUSIONS
======================================================================
🏠 Same-Domain Transfer (NY Housing):
  ✅ Neural features improve performance by 73.5%
  🧠 The network learned useful representations beyond original features

📈 Cross-Domain Transfer (Housing → Stock):
  🎉 Significant transfer! Neural features improve by 16.1%
  🔄 Learned representations generalize across domains

✅ Feature-transfer experiment complete!
📝 Key insight: Transfer learning reveals whether neural networks learn
   generalizable representations beyond task-specific feature engineering
```