Periwinkle Trading

Team: Henry Hammer, Hoang Viet Chu (PZ), Rosy Chen, Ryan Butler (TL), Tommy Binh Lu (CMC)

Liaison: Scott Smallwood, CEO
Advisor: Professor Weiqing Gu

About Periwinkle Trading

- A remote, statistical arbitrage, systematic trading firm
- Use methods of analysis on commercial data
- Name comes from the Fibonacci-like spirals of a Periwinkle snail’s shell

How does Periwinkle Use Statistical Arbitrage

- Use market flow and data to make mathematically positive expected value trades
- Mostly short holding-period trades, but not high frequency
- Don’t hold opinions on major events nor carry long-term stock positions

Project Goals

Recreate Periwinkle Architecture

Implement Modularity and Automation for Experiments

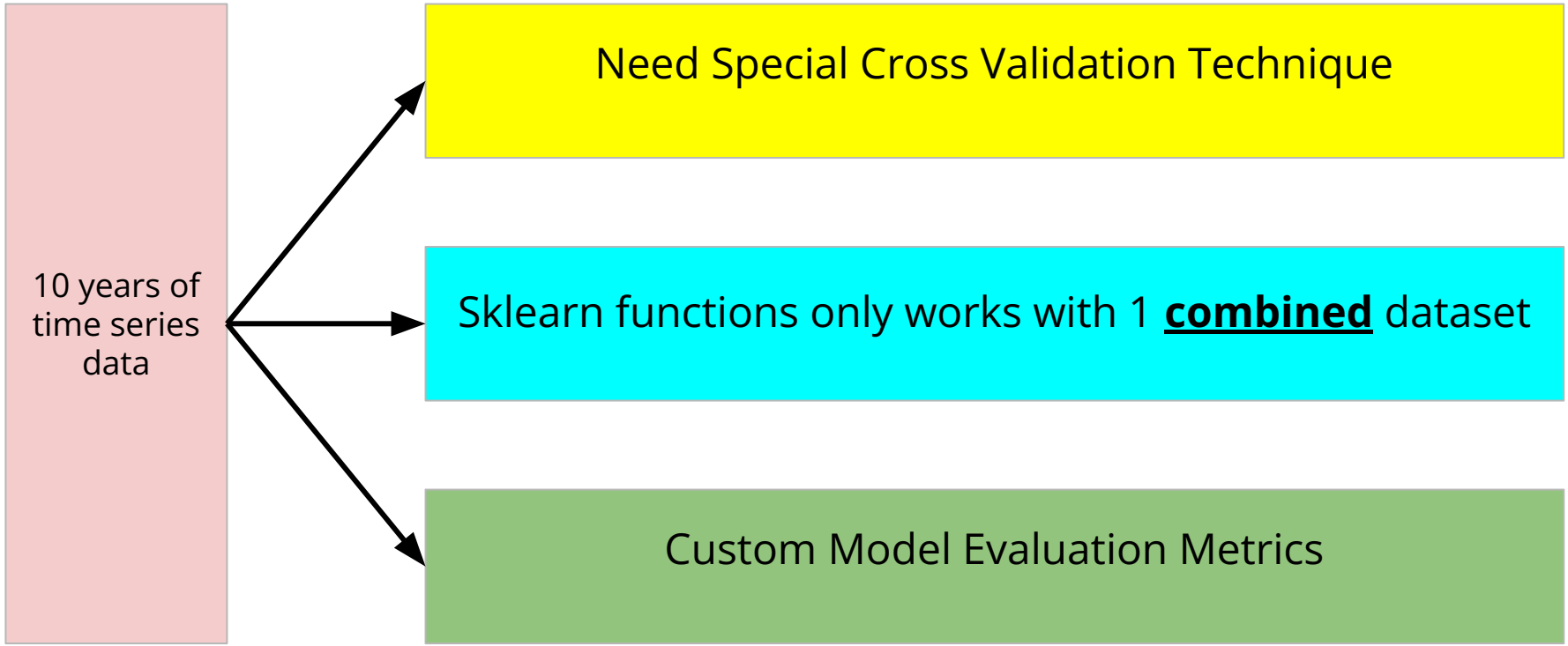
Test New Ideas

Periwinkle TRADING LLC

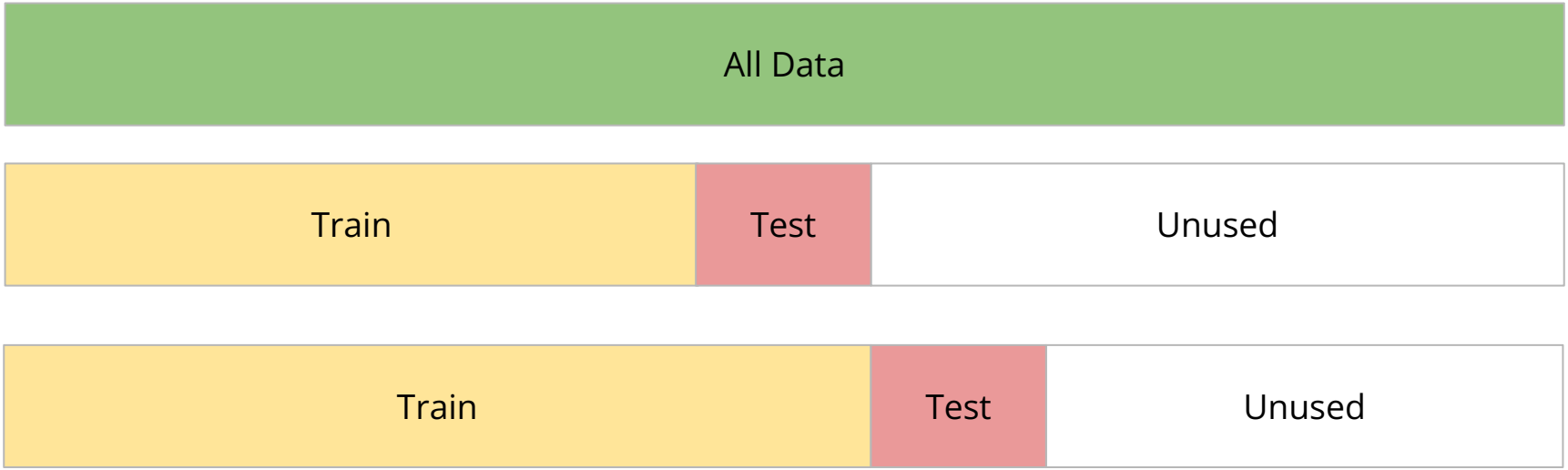
HARVEY MUDD COLLEGE

What makes our project special?

Project Specific Challenges - The Data



Time Series Cross Validation



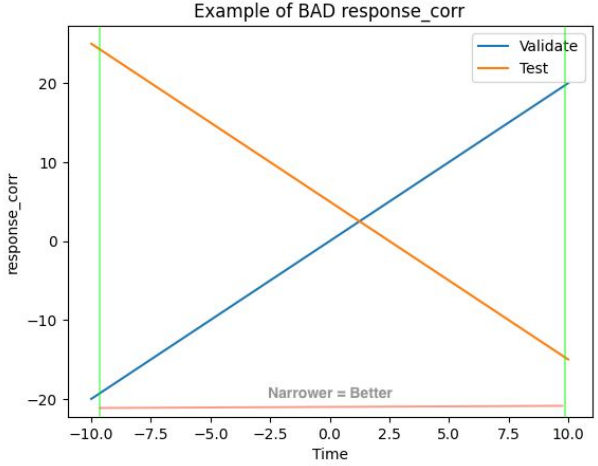
Periwinkle TRADING LLC

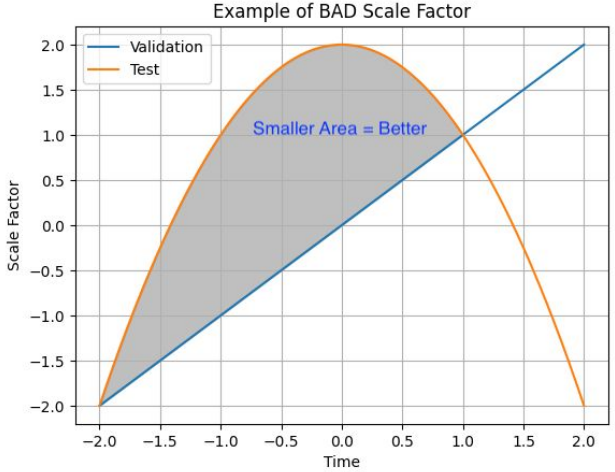
HARVEY MUDD COLLEGE

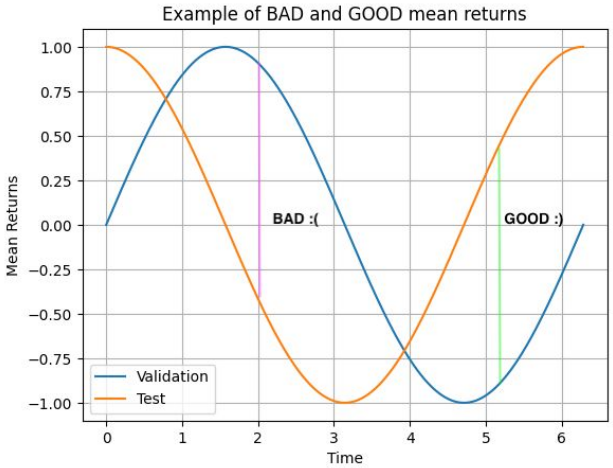
Evaluation Metrics

Customized Evaluation Metrics

- Response Correlation
  - Correlation between predicted and actual y
- Scale Factor
  - How much we need to multiple our prediction to match actual
- Mean Return: `np.mean(np.abs(actual_y) * (np.sign(actual_y) * np.sign(predicted_y)))`
  - How much money we gained







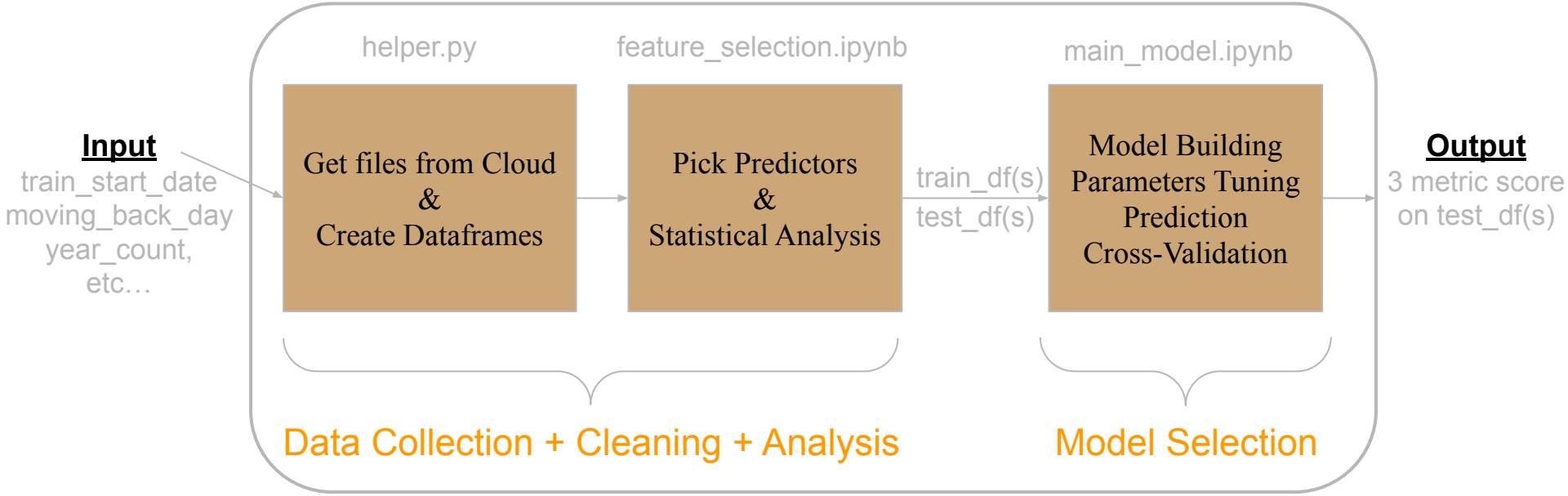
Periwinkle TRADING LLC

HARVEY MUDD COLLEGE

The Codebase

Automation

A pipeline for running experiments

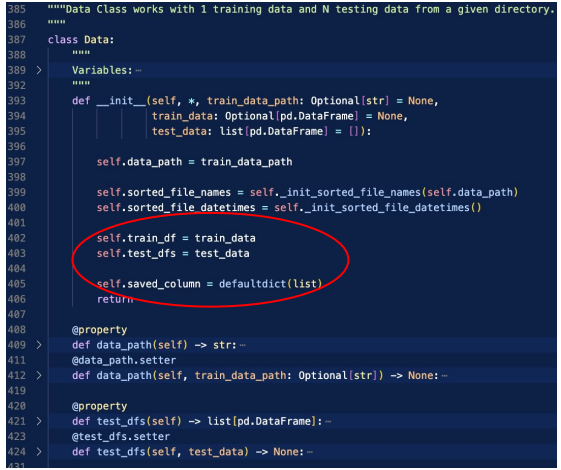


Black Box (require ideas)

Object-Oriented Design

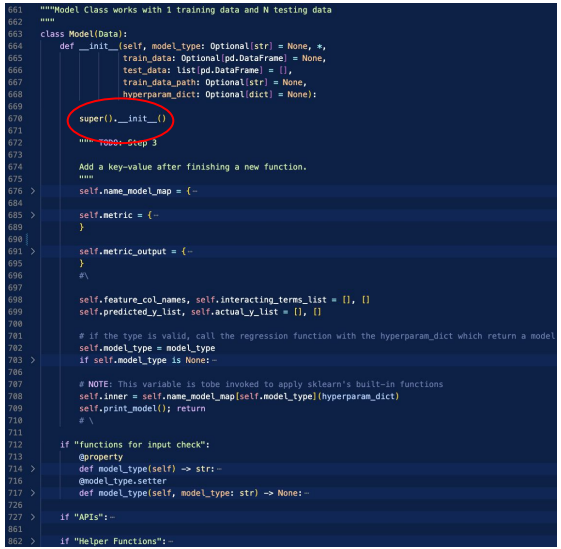
```
func extract_features_to_file
func stepwise_selection
func hypothesis_test_features
func scatter_lot
func histogram
func boxplot
func classification_validation_...
func validation_plot
class Data
class Model
```

Helper EcoSystem



```
"""Data Class works with 1 training data and N testing data from a given directory.
"""
class Data:
    """
    Variables:…
    """
    def __init__(self, *, train_data_path: Optional[str] = None,
                          train_data: Optional[pd.DataFrame] = None,
                          test_data: list[pd.DataFrame] = []):

        self.data_path = train_data_path

        self.sorted_file_names = self._init_sorted_file_names(self.data_path)
        self.sorted_file_datetimes = self._init_sorted_file_datetimes()

        self.train_df = train_data
        self.test_dfs = test_data

        self.saved_column = defaultdict(list)
        return

    @property
    def data_path(self) -> str:…
    @data_path.setter
    def data_path(self, train_data_path: Optional[str]) -> None:…

    @property
    def test_dfs(self) -> list[pd.DataFrame]:…
    @test_dfs.setter
    def test_dfs(self, test_data) -> None:…
```

Data Class
(272 lines)



```
"""Model Class works with 1 training data and N testing data
"""
class Model(Data):
    def __init__(self, model_type: Optional[str] = None, *,
                       train_data: Optional[pd.DataFrame] = None,
                       test_data: list[pd.DataFrame] = [],
                       train_data_path: Optional[str] = None,
                       hyperparam_dict: Optional[dict] = None):

        super().__init__()

        """ TODO: Step 3

        Add a key-value after finishing a new function.
        """
        self.name_model_map = {…

        self.metric = {…
        }

        self.metric_output = {…
        }
        #\

        self.feature_col_names, self.interacting_terms_list = [], []
        self.predicted_y_list, self.actual_y_list = [], []

        # if the type is valid, call the regression function with the hyperparam_dict which return a model
        self.model_type = model_type
        if self.model_type is None:…

        # NOTE: This variable is tobe invoked to apply sklearn's built-in functions
        self.inner = self.name_model_map[self.model_type](hyperparam_dict)
        self.print_model(); return
        # \

    if "functions for input check":
        @property
        def model_type(self) -> str:…
        @model_type.setter
        def model_type(self, model_type: str) -> None:…

    if "APIs":…

    if "Helper Functions":…
```

Model Class
(376 lines) - inherit Data

- Saves data in data objects:
  => Easy to test and build new models using saved data
  => Easy to add new data files to trained model

Example Experiment

```
if "Real-time, Continuous Development":
    importlib.reload(helper)

if "Init Model":
    model1 = helper.Model(f"{model_name}", hyperparam_dict = my_hyperparam_dict)

if "Train":
    model1.train(data_instance.train_df, feature_col_names=FEATURES)

if "Test":
    model1.test(data_instance.test_dfs)

if "Validate":
    helper.validation_plot(data_instance, model1, 10, TEST_DATE,
                           train_data_count= 30, data_path=FILE_PATH,
                           forward_dayCount = 15, features = FEATURES)
```

Pipeline: initialize custom model → predict (train) → test → validate

Periwinkle TRADING LLC

HARVEY MUDD COLLEGE

Data and Strategies

Explore Data

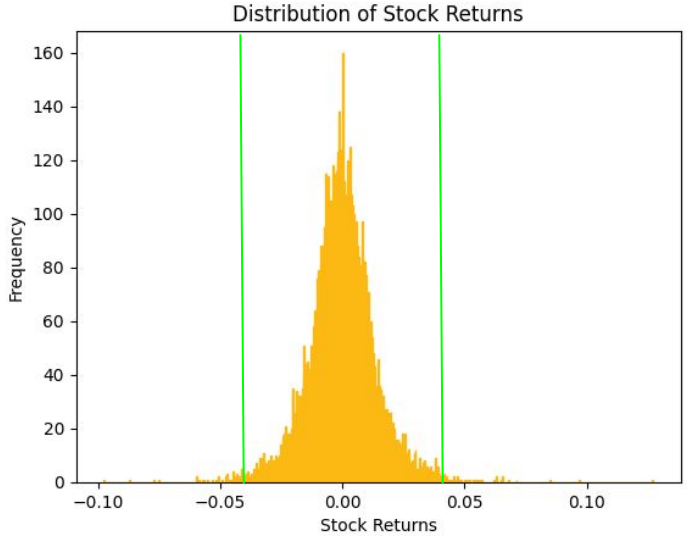
=> Target / Response Variable = 1-day stock returns (date randomly chosen).

1. **0** null values and all data is type **float**.
2. **Removed** duplicate observations and features = linear combination of others.

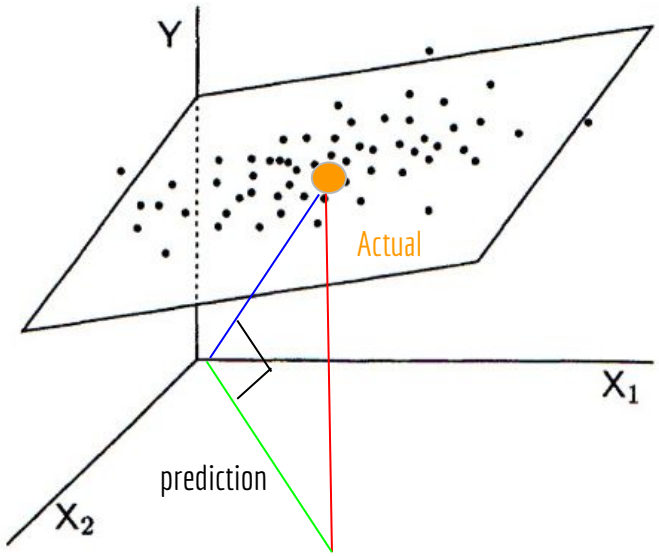
Interesting facts :)

a. -1 <= Stock Returns <= 1
b. Stock Returns ~ Normal(mean = 0)
c. Low Standard Deviation

```
Standard Deviation of Training Data: tonight    0.0138
dtype: float64
Standard Deviation of Testing Data: tonight     0.01479
dtype: float64
```



Strategy 1: With *all* features, Stock Returns follows the same path



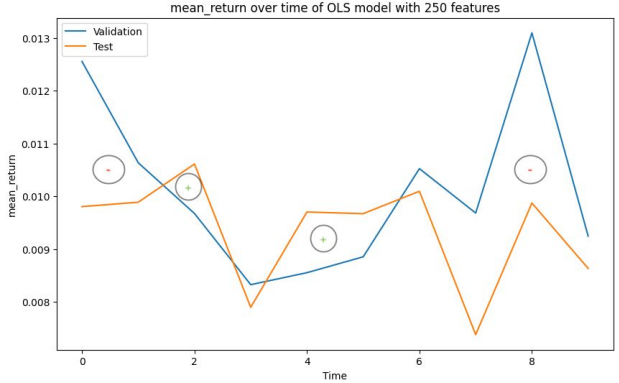
X_1	X_2
2	27

Linear Regression

Objective: Mean Squared Error

$$\text{Predicted Stock Return } = X(X^T X)^{-1} X^T \times (\text{Actual Stock Return})$$

Strategy 1 (continued): analysis for next steps



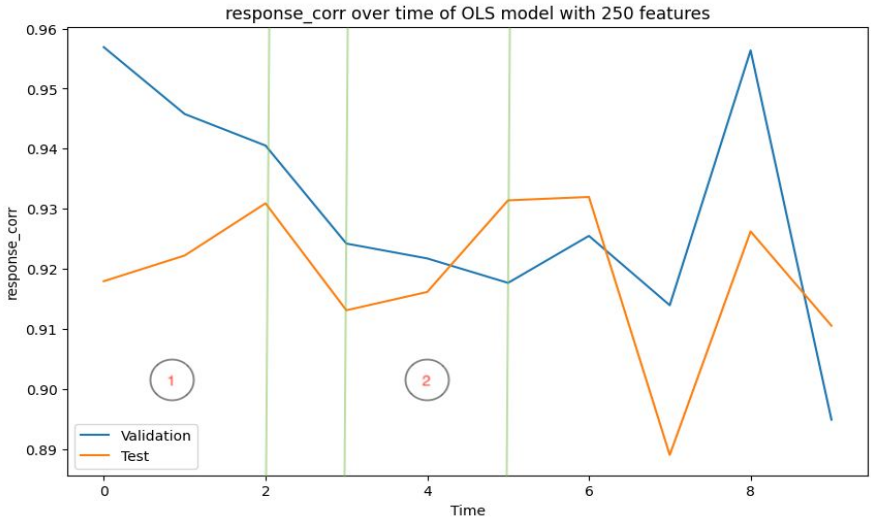
=> Overfit (response_corr, scale_factor)

Underfit (mean_returns)

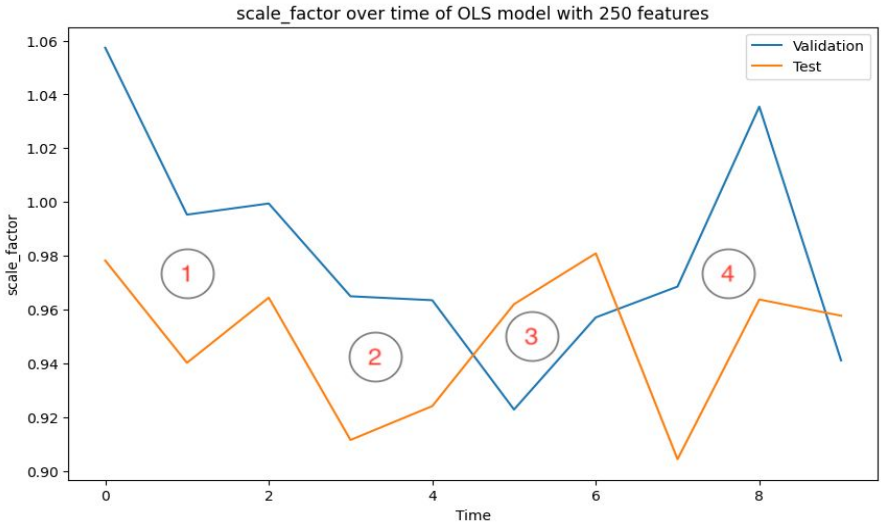
Strategy 2: Reduce features, then apply **Strategy 1**

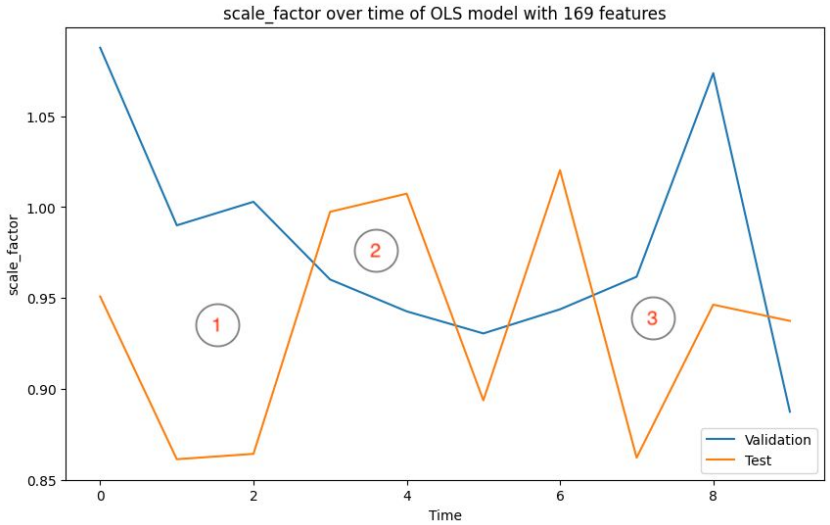
"Raise the bar": set higher objective => filter weaker features out.

- Objective: minimize **MSE** + $(0 \leq \alpha \leq 1) \times \sum_{\text{feature}} \|\beta_{i \in \{1,2,\ldots,n\}}\|$



Strategy 2 (continued): Output and Insights



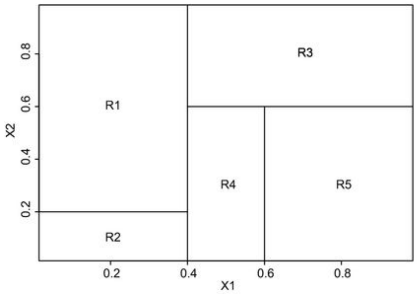


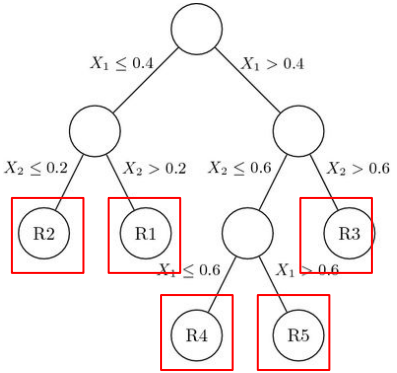
1. “Raise the bar” **reduces** accuracy when predicting **value**(stock return)
2. “Raise the bar” **improves** accuracy when predicting **sign**(stock return)
3. There exists observations that make the stock return **switch direction**.

Strategy 3: Divide and Conquer

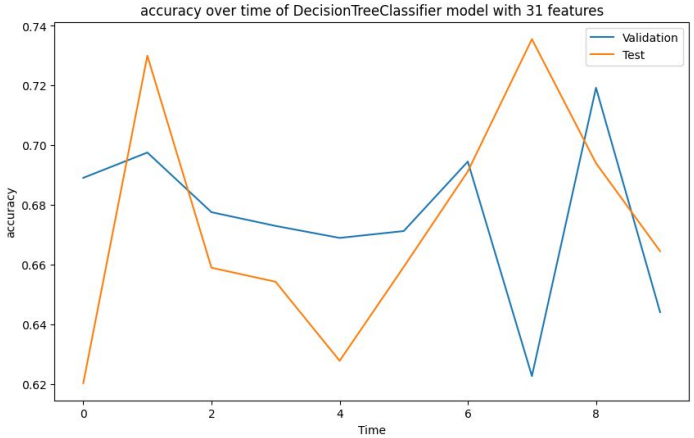
- (3) => **FORGET** features, **FOCUS ON** observations to find its **limit point**.
- Observation limit point = decision tree split point :
  => after this observation, the direction of Stock Returns changes.

- (2) => Predicting label accurately first => Objective: > 50% correct
  1. Transformed (temporarily) response to {-1, 0, 1}
  2. Find "limit observations" that the label changes
     3. For all observations:
        a. Assign it = "observation limit point",
        b. Build Classification Tree to the fullest based on *mean_allChild(Ratio(Correct, Total)^2) > 0.5*
        c. **"Raise The Bar"** with alpha from 0.01 to 1 => pruning
  4. Transfer hyperparameters to Prediction Tree, then apply **Strategy 1**
     => General Stepwise Additive Model. Or just Boosting Tree!






```
kwarg1 = {"random_state": 0,
          "booster": "gbtree",
          "reg_lambda": hyperparam_dict["ccp_alpha"],
          "max_depth": hyperparam_dict["max_depth"],
          "max_leaves": hyperparam_dict["max_leaf_nodes"],
          "random_state": hyperparam_dict["random_state"],
          "eval_metric": sklearn.metrics.mean_absolute_error,
          "learning_rate": 0.01, # cuz hyperparam_dict["ccp_alpha"] has 2 decimals
          }
model1 = helper.Model("XGBoost", hyperparam_dict = kwarg1)
model1.train(boost_data.train_df, feature_col_names = FEATURES)
model1.test(boost_data.test_dfs)
```



Strategy 3 (continued): Optimization

First Experiment:

```
helper.validation_plot(rf_regression_data, model3, 10, TEST_
                                    data_path=FILE_PATH, forward_dayCount
✓ 117m 4.7s
```

1. Function TreeBuilding(data, features, alpha)
   For all observations (*):
   - Pick that observation as the “observation limit”
   - **Build Classification Tree to the fullest**

   => Recursion to (*) => exponential runtime.
   => DFS with memoization:
   => Time Complexity = (MaxTree Size x No. observations)

2. During Cross Validation: Time Complexity = TreeBuilding x No. CV

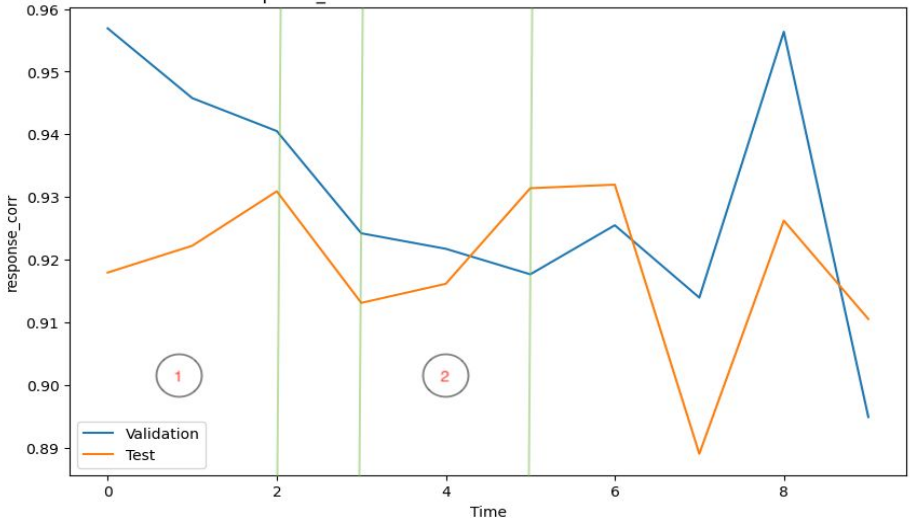
+ **OOP** + **Deque(**test_dfs**) =>** Amortized (TreeBuilding) runtime

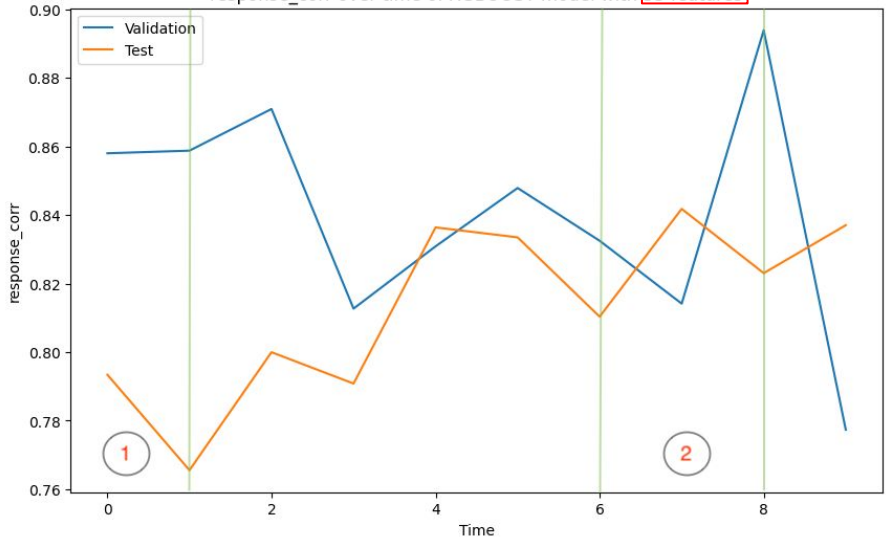
```
def optimized_tree_building(data, features, alpha):

    clf = Model('DecisionTreeClassifier')
    clf.train(data.train_df, feature_col_names=features)
    clf.test(data.test_dfs)

    costs = [0] * n_nodes; pruned = [False] * n_nodes

    if "Expand the Tree until New loss > Old loss":…

    if "The cost of the root is the total cost of the tree at index 0":

    if "Prune the tree by setting the children of pruned nodes to -1":…
```

```
filtered_features = filter_features(FEATURES, feature_important_scores)
helper.validation_plot(rf_data, transform_func, model2, 10, TEST_DATE, train_data_count= 30,
                       data_path=FILE_PATH, forward_dayCount = 15, features = filtered_features)
✓ 3m 44.0s
```

Strategy 3: Output

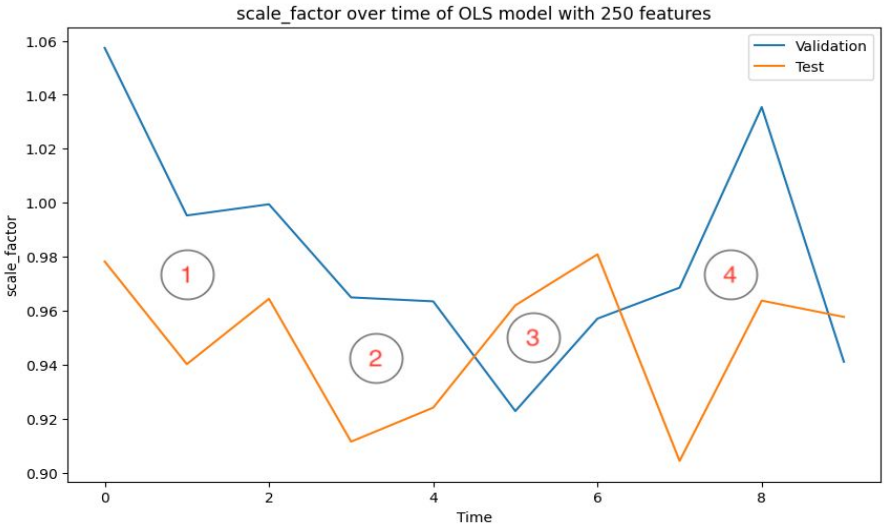
1. Response Correlation (how consistent is our price movement prediction)

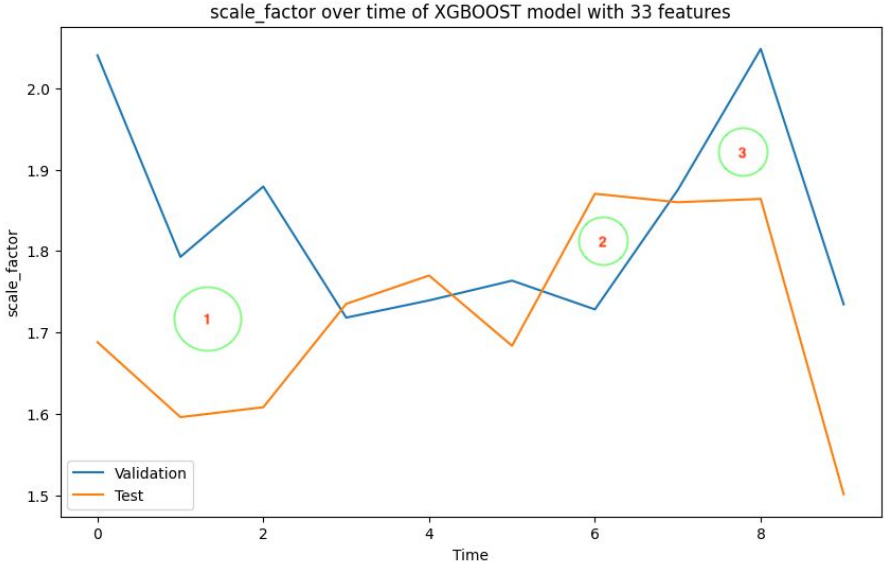




Strategy 3: Output

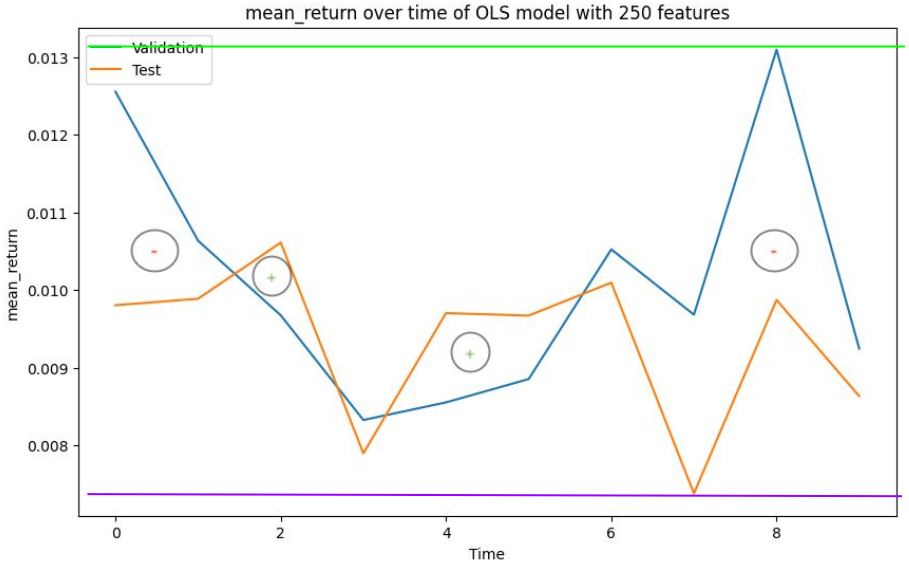
2. Scale Factor (how over-scaled / under-scaled is our predicted price)

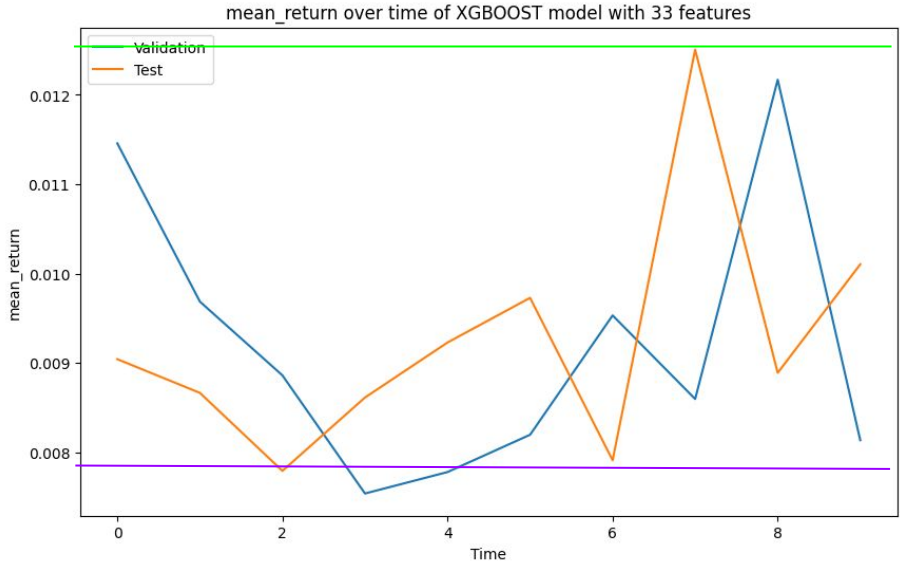




Strategy 3: Output

3. Mean Returns (what will be our relative percentage profit if we put money in?)

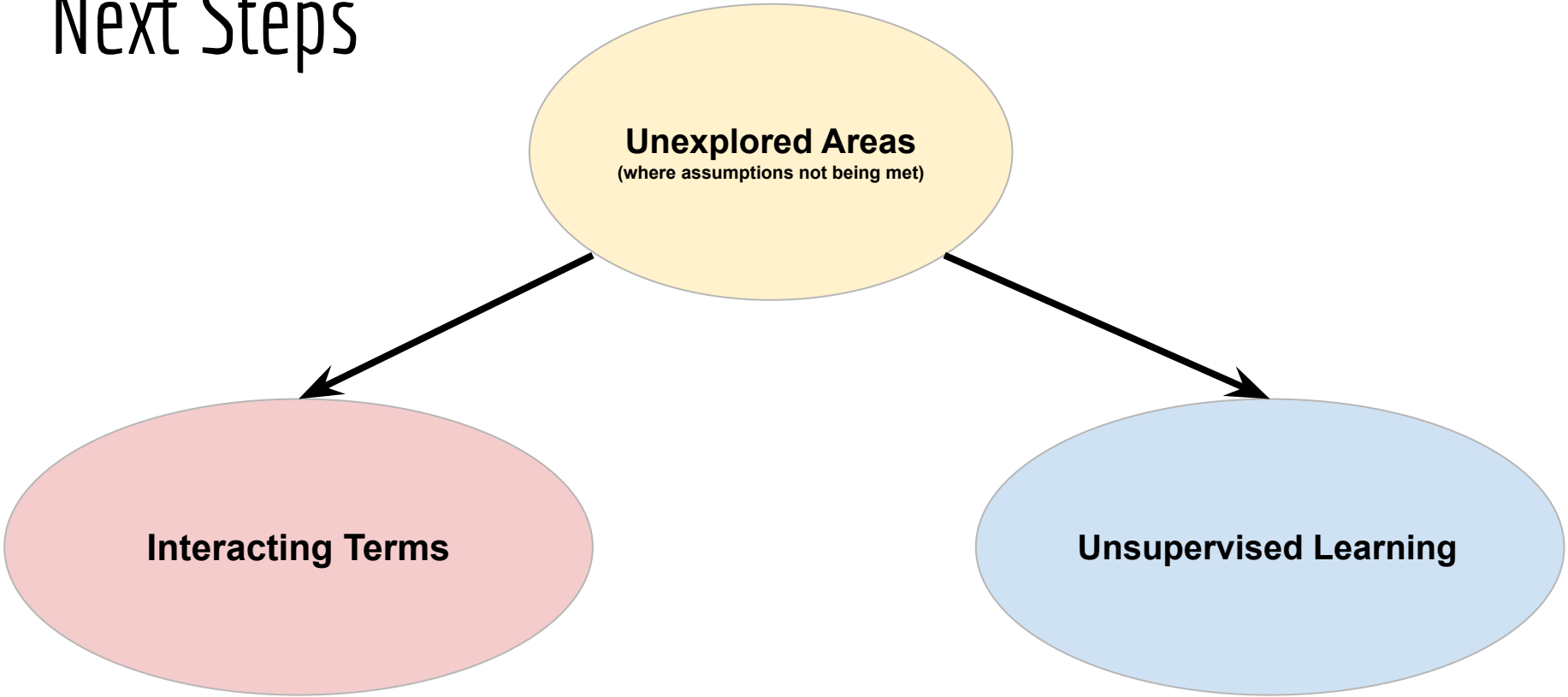




=> Solved Overfitting and Partially Solved Underfitting

Next Steps

Next Steps


Unexplored Areas
(where assumptions not being met)
Interacting Terms
Unsupervised Learning

Periwinkle TRADING LLC

HARVEY MUDD COLLEGE

Thank you for listening!