Financial Market Performance Forecasting

INFO 523 - Fall 2025 Final Project
Team 5 Members:
Nicholas Tyler, John Moran, Zuleima Cota

University of Arizona
College of Information Science

Intro / Problem Statement

Background

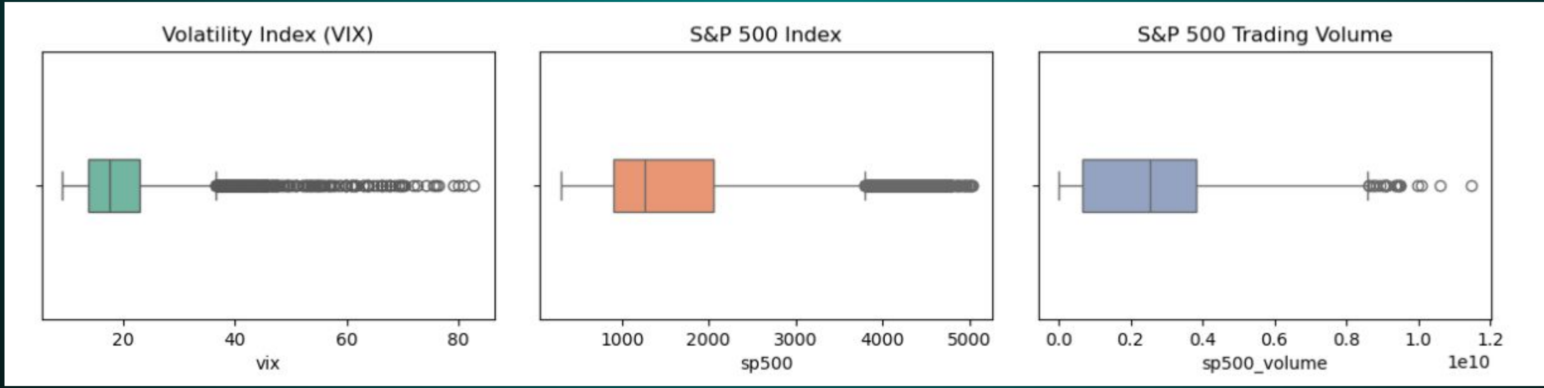
- Analyzing financial market data can provide insights that help inform sound investment strategies and mitigate risk for businesses and individual investors.

- The goal of this analysis was to explore long-term volatility patterns and conduct stock market price forecasting.

Research Questions

- Can we predict the Volatility Index within the financial markets based on various economic factors (i.e. unemployment rates, U.S. Treasury 3-Month Bond Yield, etc.)?

- What insights can historical stock market data offer to forecast future price movements and assess market volatility?



Data Sources

Daily Stock Data

- Our primary data set captures data on key financial market metrics over 34 years (1990 - 2024)

- Contains relevant information related to volume, macroeconomic indicators, volatility index, and uncertainty metrics

- Compiled from a variety of historical records including, the Chicago Board Option Exchange, Yahoo Finance, Bureau of Economic Analysis, Federal Reserve, Economic Policy Uncertainty Index, and the Global Policy Uncertainty Database

	dt	vix	sp500	sp500_volume	djia	djia_volume	hsi	ads	us3m	joblessness	epu	GPRD	prev_day
0	1990-01-03	18.19	358.760010	192330000.0	2809.73	23.62	2858.699951	-0.229917	7.89	3	100.359178	75.408051	359.690002
1	1990-01-04	19.22	355.670013	177000000.0	2796.08	24.37	2868.000000	-0.246065	7.84	3	100.359178	56.085804	358.760010
2	1990-01-05	20.11	352.200012	158530000.0	2773.25	20.29	2839.899902	-0.260393	7.79	3	100.359178	63.847675	355.670013
3	1990-01-08	20.26	353.790009	140110000.0	2794.37	16.61	2816.000000	-0.291750	7.79	3	100.359178	102.841156	352.200012

https://www.kaggle.com/datasets/shiveshprakash/34-year-daily-stock-data

Data Sources

Unemployment Rates

- Our secondary data set contains monthly unemployment rates over 34 years (1990 - 2024)
- This data set was used in conjunction with the Daily Stock Market Data to provide additional financial uncertainty metrics

	Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	1990	5.4	5.3	5.2	5.4	5.4	5.2	5.5	5.7	5.9	5.9	6.2	6.3
1	1991	6.4	6.6	6.8	6.7	6.9	6.9	6.8	6.9	6.9	7.0	7.0	7.3
2	1992	7.3	7.4	7.4	7.4	7.6	7.8	7.7	7.6	7.6	7.3	7.4	7.4
3	1993	7.3	7.1	7.0	7.1	7.1	7.0	6.9	6.8	6.7	6.8	6.6	6.5
4	1994	6.6	6.6	6.5	6.4	6.1	6.1	6.1	6.0	5.9	5.8	5.6	5.5

https://data.bls.gov/timeseries/LNS14000000

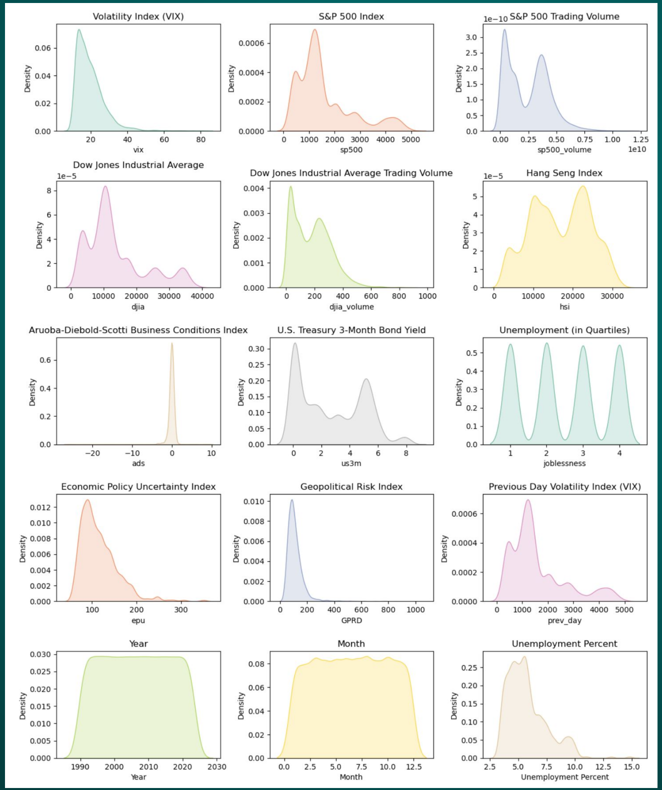
Exploratory Data Analysis

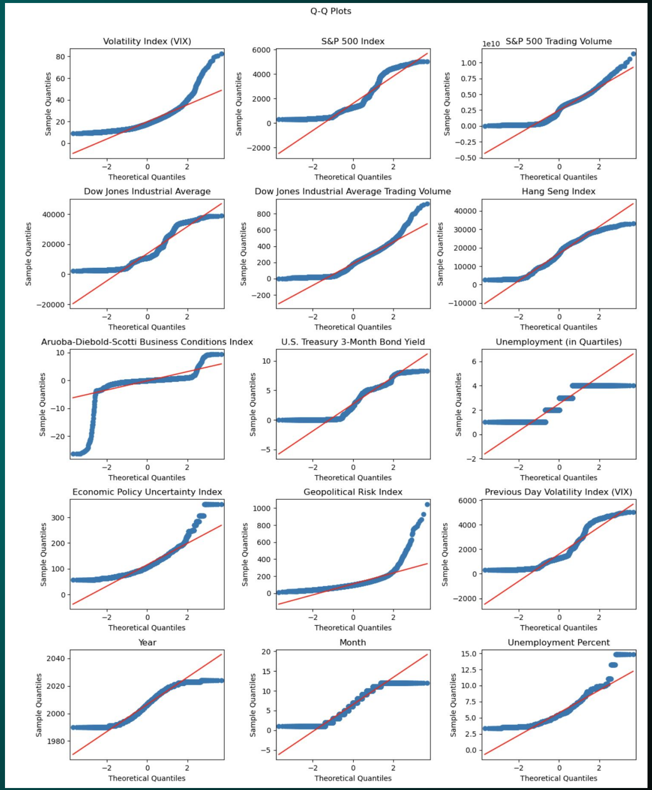
Positively Skewed Features:

- Volatility Index (VIX)
- S&P 500 Index
- S&P 500 Index Trading Volume
- Dow Jones Industrial Average
- Dow Jones Industrial Average Trading Volume
- Economic Policy Uncertainty Index
- Geopolitical Risk Index
- Previous Day VIX
- Unemployment Percent

Multimodal Features:

- S&P 500 Trading Volume
- Hang Seng Index
- U.S. Treasury 3-Month Bond Yield
- Unemployment (in Quartiles)





Question #1: Can we predict the Volatility Index within the financial markets based on various economic factors?

Models Used:

- Ridge Regression
- Random Forest Regression
- Lasso Regression
- Gradient Boost Regression

Model Evaluation: Ridge Regression

- Three Ridge models were created and trained using the batch_ridge() function
- Model 1 - Unemployment Rate Performance
  - Mean Squared Error: 53.525
  - R-Squared: 0.097
- Model 2 - Bond Yield Rate Performance
  - Mean Squared Error: 58.619
  - R-Squared: 0.0115
- Model 3 - Unemployment Rate & Bond Yield Rate Performance
  - Mean Squared Error: 53.178
  - R-Squared: 0.103

```python
def batch_ridge(models: dict[Ridge, tuple[np.array,  np.array, str]], y_train, y_test):
    '''
    Runs batchs of Ridge Regression models

    Parameters
    ----------
    models : dict
        Dictionary that contains the model as a key and a tuple for the value. The tuple
        has 3 items that represent the following (X_train, X_test descriptor)
    y_train : array-like
      Train target data
    y_test : array-like
      Test target data
    '''
    print('Begining Batch Model Traing')
    print('------------------------------------------------')
    for model, params in models.items():
        X_train = params[0]
        X_test = params[1]
        desc = params[2]
        print(f'Starting training for {desc}')
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        print(f'Model results for {desc}')
        print("Coefficients:", model.coef_)
        print("Intercept:", model.intercept_)
        mse = mean_squared_error(y_test, y_pred)
        print("MSE:", mse)
        print("R^2:", model.score(X_test, y_test))
        print('--------------------------------------------')
```

```python
unemployment_ridge = Ridge()
bond_ridge = Ridge()
bond_unemployment_ridge = Ridge()
print(y_train)
batch_args = {
    unemployment_ridge: (X_unemployment_train, X_unemployment_test, 'Unemployment Ridge Model'),
    bond_ridge: (X_bond_train, X_bond_test, 'Bond Yield Ridge Model'),
    bond_unemployment_ridge : (X_bond_jobless_train, X_bond_jobless_test, 'Bond Yield Ridge & Unemployment Model')
}
mlt.batch_ridge(batch_args, y_train, y_test)
```

```
[12.49 11.43 17.39 ... 17.43 13.1  16.36]
Begining Batch Model Traing
------------------------------------------------
Starting training for Unemployment Ridge Model
Model results for Unemployment Ridge Model
Coefficients: [2.41130135]
Intercept: 19.59399618216382
MSE: 53.52527358425855
R^2: 0.09736849352362653
--------------------------------------------
Starting training for Bond Yield Ridge Model
Model results for Bond Yield Ridge Model
Coefficients: [-0.82344292]
Intercept: 19.574020376417163
MSE: 58.6193196701746
R^2: 0.011464280715590913
--------------------------------------------
Starting training for Bond Yield Ridge & Unemployment Model
Model results for Bond Yield Ridge & Unemployment Model
Coefficients: [ 2.34762818 -0.55014837]
Intercept: 19.587718685495428
MSE: 53.17808010982467
R^2: 0.10322344293128705
```

Model Evaluation: Random Forest Regression

- This model was trained using the random_forest() function
- All variables excluding date were included in our model training
- Randomized Search Cross-Validation was used to find the best model fit with the following hyperparameters
    - n_estimators: 200
    - min_samples_split: 5
    - max_features: log2
    - max_depth: None
    - bootstrap: False
- The best model achieved the following performance
    - Mean Squared Error: 2.096
    - R-Squared: 0.965

```
# Splitting the Data into Training and Testing Sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
mlt.random_forest(x_train, x_test, y_train, y_test)
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits
Best parameters found:
{'n_estimators': 200, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': None, 'bootstrap': False}
Random Forest MSE: 2.0961
Random Forest R^2: 0.9647
```

```
def random_forest(x_train, x_test, y_train, y_test, param_dist=None, cv=5, n_iter=25, random_state=42):
    """
    Train a Random Forest regressor with optional hyperparameter tuning using RandomizedSearchCV.

    Parameters
    ----------
    x_train : np.ndarray or pd.DataFrame
        Training features.
    x_test : np.ndarray or pd.DataFrame
        Test features.
    y_train : np.ndarray or pd.Series
        Training target.
    y_test : np.ndarray or pd.Series
        Test target.
    param_dist : dict, optional
        Hyperparameter search space for RandomizedSearchCV. If None, a default
        distribution is used.
    cv : int, optional
        Number of cross-validation folds for hyperparameter search.
    n_iter : int, optional
        Number of parameter settings sampled in RandomizedSearchCV.
    random_state : int, optional
        Random seed for reproducibility.

    Returns
    -------
    best_model : RandomForestRegressor
        Best estimator found by RandomizedSearchCV (or the default model if no search).
    y_pred : np.ndarray
        Predictions on the test set.
    """
    # Base model
    base_rf = RandomForestRegressor(random_state=random_state)

    # Default hyperparameter search space if none provided
    if param_dist is None:
        param_dist = {
            "n_estimators": [100, 200, 300, 500],
            "max_depth": [None, 5, 10, 20, 30],
            "min_samples_split": [2, 5, 10],
            "min_samples_leaf": [1, 2, 4],
            "max_features": ["sqrt", "log2", 0.8],
            "bootstrap": [True, False],
        }

    # Hyperparameter tuning
    search = RandomizedSearchCV(
        estimator=base_rf,
        param_distributions=param_dist,
        n_iter=n_iter,
        cv=cv,
        scoring="neg_mean_squared_error",
        random_state=random_state,
        n_jobs=-1,
        verbose=1
    )

   best_model = search.best_estimator_
   print("Best parameters found:")
   print(search.best_params_)

   # Evaluate on test set
   y_pred = best_model.predict(x_test)
   rf_mse = mean_squared_error(y_test, y_pred)
   rf_r2 = r2_score(y_test, y_pred)

   print(f"Random Forest MSE: {rf_mse:.4f}")
   print(f"Random Forest R^2: {rf_r2:.4f}")
```

Model Evaluation: Lasso Regression

- This model was trained using the lasso_regression() and lasso_hyperparameter_tuning() functions
- All variables excluding date were included in our model training
- Lasso Cross Validation was used to find the best Alpha value
- The following hyperparameters were determined to provide our model with the best fit:
  - Alpha: 0.001
- This model performed as follows:
  - Mean Squared Error: 0.0479
  - R-Squared: 0.5605

```python
# Dropping the Date Columns
stock_data_final_no_date = stock_data_final_standardized.drop(columns = ['dt'])

print('Original Lasso Regression Results:')
print(lr.lasso_regression(stock_data_final_no_date, 'vix', 1.0, 42),'\n')


print('Lasso Regression Results with Hyperparameter Tuning:')
print(lr.lasso_hyperparameter_tuning(stock_data_final_no_date, 'vix', 5, 100000, 42))
```

```
Original Lasso Regression Results:
Mean Squared Error: 0.10806197639498576
R-Squared: -0.00017192695750201104

Lasso Regression Results with Hyperparameter Tuning:
Optimal alpha: 0.0001
Mean Squared Error: 0.0474877605402655
R-Squared: 0.560475140739165
```

```python
def lasso_hyperparameter_tuning(df, y, cv, max_iter, random_state):
    '''
    This function performs hyperparameter tuning on the lasso model,
    fits the model with the new parameters, and returns the Mean Squared
    Error and R-Squared values.

    Parameters
    -------------
    df (pd.DataFrame):
        The DataFrame that the mdoel will be trained on. This DataFrame should
        only contain numeric values.
    y (pd.Series):
        The response variable from the DataFrame.
    cv (int):
        The number of cross-validation folds for the Lasso Regression model.
    max_iter (int):
        The max number of iterations.
    random_state (int):
        The seed set for reproducibility within our model.

    Returns
    -------------
    results (str):
        The Mean-Squared and R-Squared results of the Lasso Regression Model.
    '''
    # Initializing a variable for the predictor variables
    X = df.drop(y, axis = 1)

    # Initializing a variable for the response variable
    y = df[y]

    # Splitting the data into training and testing datasets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = random_state)

    # Defining a range of alpha values for Lasso
    alphas = np.logspace(-4, 2, 100)

    # Initializing LassoCV
    lasso_cv = LassoCV(alphas = alphas, cv = cv, max_iter = max_iter, random_state = 42)

    # Fitting the Model
    lasso_cv.fit(X_train, y_train)

    # Returning the Results for the Optimal alpha
    print(f'Optimal alpha: {lasso_cv.alpha_}')

    # Re-Initializing the Model and Fitting it with the Optimal Alpha
    best_lasso = Lasso(alpha = lasso_cv.alpha_, max_iter = max_iter)
    best_lasso.fit(X_train, y_train)

    # Making New Predictions
    y_pred_best_lasso = best_lasso.predict(X_test)

    # Calculating the Mean Squared Error
    mse_best = mean_squared_error(y_test, y_pred_best_lasso)

    # Calculating the R-Squared Value
    r2_best = r2_score(y_test, y_pred_best_lasso)

    # Initializing a Variable for the MSE and R-Squared Results
    results = f'Mean Squared Error: {mse_best}\nR-Squared: {r2_best}'
```

Model Evaluation: Gradient Boost Regression

- Two Gradient Boost Regression Models were trained: one including Principal Component Analysis (PCA) to reduce dimensionality and one excluding PCA
  - The model excluding PCA outperformed the model including PCA, so all results mentioned in this presentation will be referring to this model
- This model was trained using the gradient_boost() and gradient_boost_hyperparameter_tuning() functions
- The following hyperparameters were determined by Grid Search Cross-Validation to provide our model with the best fit:
  - Learning Rate: 0.1
  - Max Depth: 7
  - Number of Estimators: 300
- This model performed as follows:
  - Mean Squared Error: 0.0040
  - R-Squared: 0.9626

```python
# Performing Gradient Boost without PCA Applied
print('Gradient Boost Regression Results without PCA Applied:\n')
print('Original Gradient Boost Regression Results:')
print(gb.gradient_boost(stock_data_final_no_date, 'vix', 100, 42), '\n')

print('Gradient Boost Regression Results with Hyperparameter Tuning:')
print(gb.gradient_boost_hyperparameter_tuning(stock_data_final_no_date, 'vix', param_grid, 5, 42))
```

```
Gradient Boost Regression Results without PCA Applied:

Original Gradient Boost Regression Results:
Mean Squared Error: 0.013756544944325256
R-Squared: 0.8726757503032148

Gradient Boost Regression Results with Hyperparameter Tuning:
Best Parameters: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 300}
Mean Squared Error: 0.0040380629143249155
R-Squared: 0.9626255478482681
```

```python
def gradient_boost_hyperparameter_tuning(df, y, param_grid, cv, random_state):
        The DataFrame that the model will be trained on. This DataFrame should
        only contain numeric values.
    y (pd.Series):
        The response variable from the DataFrame.
    param_grid (dict):
        The dictionary that holds the values for Grid Search Cross-Validation.
    cv (int):
        The number of cross-validation folds for the Gradient Boosting Regression model.
    random_state (int):
        The seed set for reproducibility within our model.

    Returns
    -------------
    results (str):
        The Mean-Squared and R-Squared results of the Gradient Boosting Regression Model.
    '''
    # Initializing a variable for the predictor variables
    X = df.drop(y, axis = 1)

    # Initializing a variable for the response variable
    y = df[y]

    # Splitting the data into training and testing datasets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

    # Initializing a variable for the gradient boost model
    gradient_boost = GradientBoostingRegressor(random_state = random_state)

    # Initializing a Variable for Grid Search
    grid_search = GridSearchCV(gradient_boost,
                               param_grid = param_grid,
                               cv = cv,
                               scoring = 'neg_mean_squared_error',
                               n_jobs = 1)

    # Fitting the Model
    grid_search.fit(X_train, y_train)

    # Re-Initializing and Fitting the Model with the new Parameters
    gradient_boost_best = gradient_boost.set_params(**grid_search.best_params_)
    gradient_boost_best.fit(X_train, y_train)

    # Returning the Best Parameters from Grid Search
    print(f'Best Parameters: {grid_search.best_params_}')

    # Making New Predictions on the Test Dataset
    y_pred_best = gradient_boost_best.predict(X_test)

    # Calculating the Mean Squared Error
    mse_lasso = mean_squared_error(y_test, y_pred_best)

    # Calcuating the R-Squared Value
    r2_lasso = r2_score(y_test, y_pred_best)

    # Initializing a Variable for the MSE and R-Squared Results
    results = f'Mean Squared Error: {mse_lasso}\nR-Squared: {r2_lasso}'

    # Returning the Results
    return results
```

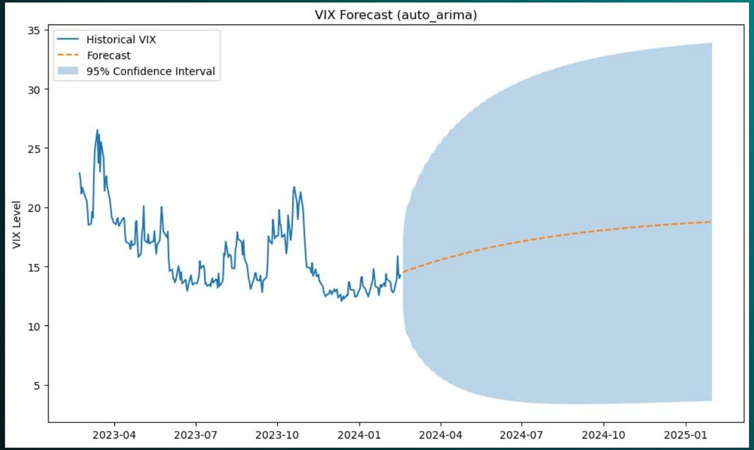
Question #2: What insights can historical stock market data offer to forecast future price movements and assess market volatility?

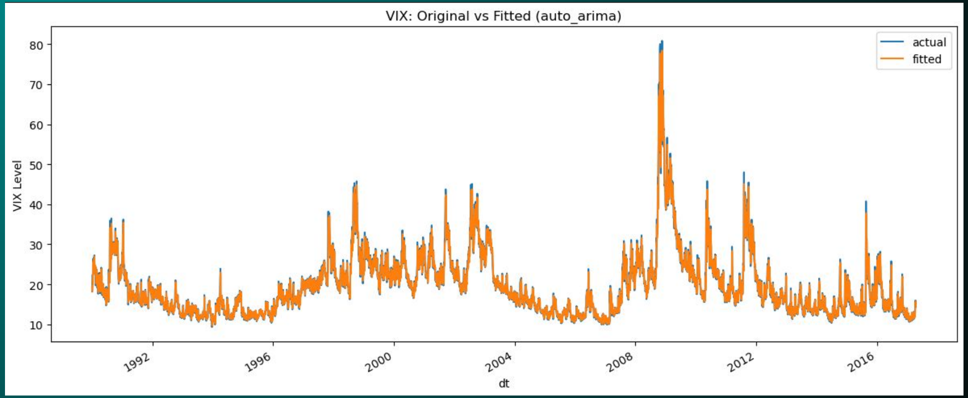
Models Used:

- ARIMA
- LSTM
- Prophet

Model Evaluation: ARIMA

- The auto_arima model was used to create VIX forecast
- The models forecasted VIX scores are shown as a smooth line with positive slope
- The models forecast 95% confidence interval is quite large, indicating high uncertainty in its predicted values
- The close alignment between fitted and actual values suggests the model is capturing short-term autocorrelation patterns, but it may also be a result of overfitting





Model Evaluation: LSTM

- All variables were included in the LSTM model
- We achieved the following performance results:
  - Mean Squared Error: 4.6914
  - R-Squared: -9.715
- Due to its subpar performance, we completely disregarded this model from our analysis

Long Short-Term Memory (LSTM)

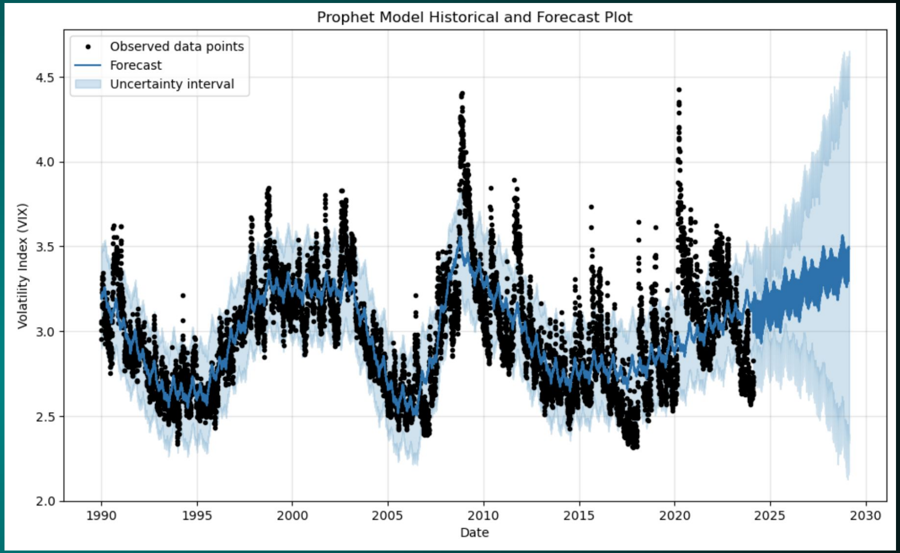
```
print(lstm.lstm_function(stock_data_final_no_date, 'vix'))
```

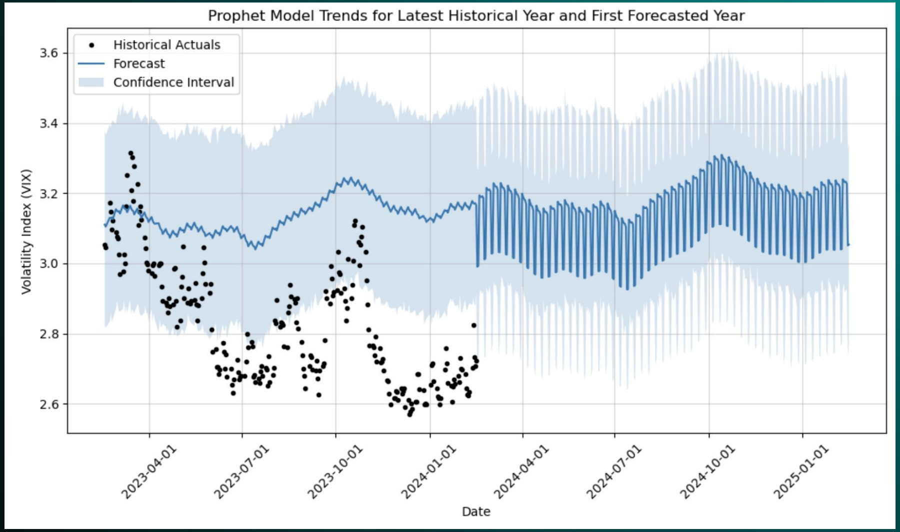
```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Mean Squared Error: 1.2614137380669403
R-Squared: -9.71472736757814
```

```python
def lstm_function(df, y):
        The Mean-Squared and R-Squared results of the LSTM model.
    '''
    # Initializing a Variable for the Predictor Columns
    predictor_cols = df.columns.drop(y)

    # Creating a Variable for the X and y Variables
    X_all = df[predictor_cols].values
    y_all = df[[y]].values

    # Creating a one-step lag: current features -> next-step target
    X = X_all[:-1]
    y = y_all[1:]

    # Splitting the Data into Training and Testing Datasets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)

    # Reshaping for LSTM
    X_train = X_train.reshape((X_train.shape[0], 1, X_train.shape[1]))
    X_test = X_test.reshape((X_test.shape[0], 1, X_test.shape[1]))

    # Creating the LSTM Model
    model = Sequential([LSTM(50, activation='relu', input_shape=(X_train.shape[1], X_train.shape[2])),
                        Dense(1)])

    # Compiling the Model
    model.compile(optimizer='adam', loss='mse')

    # Fitting the Model
    model.fit(X_train, y_train, epochs=100, batch_size=32,
                        validation_split=0.2, verbose=0)

    # Predicting the Model
    y_pred = model.predict(X_test)

    # Calculating the Mean Squared Error and R-Squared Value
    mse = mean_squared_error(y_test.ravel(), y_pred.ravel())

    # Calculating the R-Squared Value
    r2_val = r2_score(y_test.ravel(), y_pred.ravel())

    # Returning the Results
    result = f'Mean Squared Error: {float(mse)}\nR-Squared: {float(r2_val)}'

    return result
```

Model Evaluation: Prophet

- Only two variables were considered for the Prophet model: the date and the Volatility Index
- This model achieved the following performance metrics:
  - Mean Squared Error: 0.0478
  - R-Squared: 0.5618
- As the forecasted date gets farther away from the historical data, the confidence interval increases drastically
- The forecasted results follow a similar pattern to what is seen from the last historical year's results






```
Mean Squared Error: 0.0478
R-Squared: 0.5618
Performance Metrics:
```

Conclusion

Question 1

- Random Forest Regressor and Gradient Boost Regression we both able to explain around 96% of the variance in VIX
- Gradient Boost Regression delivered the highest performance overall
- This suggest Gradient Boost Regression effectively captures the nonlinear patterns present in market volatility

Question 2

- Prophet performed the best with
- These metrics indicate that Prophet captured a meaningful portion of VIX's temporal structure
- Although Prophet did not fully explain the variance of VIX, it's performance suggest it can model short-term movements more effectively than traditional linear or strictly autoregressive models