# Applied Machine Learning to Predict Stock Price

## 1. Introduction

### 1.1 Overview Scope

In the realm of financial markets, Machine Learning (ML) has gained significant traction for its ability to analyze and predict stock prices. Our project is to develop an application of ML in predicting stock prices in a timeframe of 15 mins for three Thai stocks including ERW, SPRC, and TISCO. The primary objective is to classify whether the stock prices are likely to rise (buy) or fall (sell) based on historical data and relevant features.

### 1.2 Motivation

- Complexity of financial markets.

- Traditional methods struggle to capture complicated patterns of stock prices.

- ML's ability to process large amounts of data for informed investment decisions.

## 2. Our Solution

We develop classification models for predicting binary class, "buy" or "sell", based on specific features. Our project develops 3 classification models to predict 3 Thai stocks price including ERW, TISCO, and SPRC separately.

### 2.1 Classification machine learning algorithm

We develop classification models by using various algorithms including SVM, KNN, Random Forest, and Logistic Regression models to classify “buy" a label of 1 and "sell" a label of -1 from predictor, historical stock price data and others. An algorithm that provides the best performance will be selected for future prediction.

### 2.2 Features or Predictor

Our project develops 3 machine models to predict 3 Thai stocks price including ERW, TISCO, and SPRC separately. However, we use the same features for each model. We select the features based on domain expert of stock trading. Mosty features are trading indicators. Let’s explore chosen indicators and others in the following lists.

#### **2.2.1 Historical Stock Prices:**

The historical stock prices and volume includes

- Open Price
- Close Price
- High Price
- Low Price

- Volume

within 15 mins timeframe are chosen as predictors.

**2.2.2) Technical Indicators:**

We use various technical indicators that capture patterns and trends in stock prices including the following lists.

**Trend Following Indicators:**

Trend-following indicators are tools that help traders identify and follow the direction of the dominant market trend. We select this indicator in this type as the following.

- *Simple Moving Average (SMA)*

In finance, the moving average (MA) is a stock indicator commonly used in technical analysis. One of MA type is a simple moving average (SMA). It is a calculation that takes the arithmetic mean of a given set of prices over a specific number of periods in the past. SMA helps to level the price data over a specified period by creating a constantly updated average price. It identifies the trend direction of a stock.

- *Moving Average Convergence Divergence (MACD)*

It identifies trend direction and potential trend changes using the difference between short-term and long-term exponential moving averages. Using this indicator, tell us where to entry or exit points in the market.

- *Parabolic SAR (Stop and Reverse)*

SAR provides potential reversal points in the direction of the trend. SAR suggests a downtrend, and uptrend.

**Momentum Indicator:**

Momentum indicators are used to determine the strength or weakness of a stock's price. Momentum measures the rate of the rise or fall of stock prices. Common momentum indicators include the relative strength index (RSI) and Stochastic Oscillator.

- *Relative Strength Index (RSI)*

It is used to measure the volatility of prices to determine whether there is an excessive buying (Overbought) or selling (Oversold) condition**.** Moreover, it can tell us potential reversals.

- *Stochastic Oscillator*

It measures the current price relative to its range over a specific period, identifying overbought or oversold conditions. It involves observing the relationship between %K and %D provides additional insights to determine buying and selling points. More details about %K and %D will be provided in topic ‘ Feature Engineering ’ .

**Volatility Indicators:**

When our stock exhibits high volatility, it means that the price is experiencing significant movements within a short period. These movements can be in either an upward or downward direction. High volatility can help us estimate potential price targets in the future.

**2.2.3) Index Price (SET50 and SET100):**

**SET 50 price :**

We utilize it for analyzing TISCO stock, which is listed in SET50. It can provides insights into the performance and trends of the top 50 stocks on the Stock Exchange of Thailand (SET).

**SET 100 price :**

We employe it for analyzing ERW and SPRC stocks, which both are listed in SET100. It can offers information on the performance and trends of the broader set of top 100 stocks on the Stock Exchange of Thailand (SET).

## 2.3 Target Variable

Our objective is to classify whether the stock prices are likely to rise (buy) or fall (sell). Therefore, we define target variables into binary class, buy the stock (1), and sell it (-1)**.**

## 2.4 Relevant Library

- *yfinance:* For fetching and managing financial data, allowing easy retrieval of historical stock prices.

- *Statistical functions (scipy.stats) :* Used for statistical analysis and modeling, providing tools for estimating and testing various statistical models.

- *Scikit-learn (sklearn):* A machine learning library providing tools for classification, regression, clustering, and more, enhancing predictive modeling capabilities.

- *ta-lib:* Technical Analysis Library for financial market data analysis, offering a wide range of technical indicators for quantitative analysis in trading strategies.

# 3. Methodology

## 3.1 Data Collection

Data for training the model is historical stock price data, which is primarily imported from **Yahoo website** by ' yfinance library' . The primary selected data is 'Open price', 'Close Price', 'Low price', 'High Price', and 'Volume'. Subsequently, they are transformed into the features for the training dataset. Moreover, the 'Index Price' which used as feature also imported in the same way. An example of the primary data is shown in the figure below.

| | Datetime | Open | High | Low | Close | Volume |
|---|---|---|---|---|---|---|
| 0 | 2024-01-05 10:00:00 | 5.30 | 5.30 | 5.25 | 5.25 | 837403 |
| 1 | 2024-01-05 10:15:00 | 5.30 | 5.30 | 5.20 | 5.30 | 1892000 |
| 2 | 2024-01-05 10:30:00 | 5.30 | 5.30 | 5.25 | 5.25 | 22711 |
| 3 | 2024-01-05 10:45:00 | 5.25 | 5.30 | 5.25 | 5.25 | 89600 |
| 4 | 2024-01-05 11:00:00 | 5.25 | 5.30 | 5.20 | 5.25 | 1745746 |
| 5 | 2024-01-05 11:15:00 | 5.25 | 5.30 | 5.20 | 5.20 | 4815501 |
| 6 | 2024-01-05 11:30:00 | 5.20 | 5.25 | 5.20 | 5.25 | 168200 |
| 7 | 2024-01-05 11:45:00 | 5.25 | 5.30 | 5.20 | 5.25 | 539268 |
| 8 | 2024-01-05 12:00:00 | 5.25 | 5.25 | 5.20 | 5.25 | 1530900 |
| 9 | 2024-01-05 12:15:00 | 5.20 | 5.25 | 5.20 | 5.25 | 270600 |
| 10 | 2024-01-05 14:15:00 | 5.25 | 5.25 | 5.20 | 5.20 | 590800 |

## 3.2 Feature Engineering

The primary data is ‘Open price’, ‘Close Price’, ‘Low price’, ‘High Price’, and ‘Volume’ of each stock. They are transformed into features as following detail.

### 3.2.1.) Simple Moving Average (SMA)

To calculate a Simple Moving Average, we leverage the Pandas ‘dataframe.rolling()’ function, which enables us to perform calculations within a rolling window. Within this window, we employ the ‘.mean()’ function to compute the mean for each interval. For our calculations, we use a 10-day window. Consequently, the 'Close Price' for the previous 10 period is computed as the mean and utilized as a feature. The code for calculating SMA is presented in the figure below.

```
#Moving Average
sprc_df['SME'] = sprc_df['Close'].rolling(window=10).mean()
sprc_df
```

### 3.2.2 Relative Strength Index (RSI)

We use the ‘talib.RSI’ function from ‘TA-Lib’ library to calculate the RSI values. We use 14 period as the lookback period for RSI calculation. The code for calculating RSI is presented in the figure below.

```
#RSI
sprc_df['RSI'] = ta.RSI(np.array(sprc_df['Close']), timeperiod =14)
sprc_df
```

### 3.2.3. Stochastic Oscillator

We calculate the stochastic oscillator using the values from our historic price data. The first step is to choose how many previous days we want to use to generate our fast signal (%k). We use 14 periods. With that, we’ll calculate the %k (fast) which we will then use to calculate the %d (slow).

This approach uses the ‘DataFrame.rolling()’ method to tell Python to use the previous 14 values for calculation. We then use the ‘max()’ and ‘min()’ functions to get our min/max for that 14 period. These values are appended to new columns in our DataFrame and made available for future calculations.

Next, we use the formula for calculating the %k as a percentage of the min/max for the previous trading periods. This will return a value between 0-100 a reflect the price relative to the preceding min/max values. Finally, we use the values generated for the %k to calculate a simple moving average over the previous 3 periods for which %k was calculated to be %D. After we got %D and %K, they were added as the features.

In the trading strategy, the stochastic oscillator also serves as a signal, indicating when to buy or sell stocks. The crossovers between %K and %D provide important signals. A buy signal occurs when %K crosses above %D, while a sell signal occurs when %K crosses below %D. Therefore, we use the difference between %K and %D as a feature.The example code is presented in the figure below.

```
#Stochastic Oscillator
# Define periods
k_period = 14
d_period = 3
# Adds a "n_high" column with max value of previous 14 periods
sprc_df['n_high'] = sprc_df['High'].rolling(k_period).max()
# Adds an "n_low" column with min value of previous 14 periods
sprc_df['n_low'] = sprc_df['Low'].rolling(k_period).min()
# Uses the min/max values to calculate the %k (as a percentage)
sprc_df['%K'] = (sprc_df['Close'] - sprc_df['n_low']) * 100 / (sprc_df['n_high'] - sprc_df['n_low'])
# Uses the %k to calculates a SMA over the past 3 values of %k
sprc_df['%D'] = sprc_df['%K'].rolling(d_period).mean()
sprc_df["Stochastic"] = sprc_df['%K'] - sprc_df['%D']

sprc_df =sprc_df[['Open', 'High', 'Low', 'Close', 'Volume', 'SME', 'RSI','%K', '%D', 'Stochastic']]
```

### 3.2.4 Volatility

We use the historical price data to calculate standard deviation, which is a measure of dispersion or volatility using the 'pandas' and 'numpy' libraries. Firstly, we calculate the returns using the 'pct_change()' function, and then compute the standard deviation of 5 period. The example code is presented in the figure below.

```
#Volatility
ret = 100 * (sprc_df.pct_change()[1:]['Close'])
sprc_df['Volatility'] = ret.rolling(5).std()
sprc_df
```

### 3.2.5. The moving average convergence divergence (MACD)

The MACD represents 3 district values, each of which are interconnected. Below are the MACD's main primary signals.

MACD – the value of an exponential moving average (EMA) subtracted from another EMA with a shorter lookback period. We use 26 period for the longer EMA and 12 for the shorter. This is referred to as the 'slow' signal line of the MACD.

Signal– the EMA of the MACD of a period shorter than the shortest period used in calculating the MACD. We use 9 periods. This is referred to as the 'fast' signal line.

Difference – The difference between the MACD – Signal line is used to represent current selling pressures in the marketplace. We use this value as a feature.

In python, we calculate the MACD using built-in functions native to 'Pandas DataFrame' objects. Pandas provides a built-in function to its DataFrame class named 'ewm()' which stands for exponentially-weighted mean. This will allow us to calculate the MACD. Consider the following code for an example:

```
#MACD

# # Calculate MACD values using the pandas_ta library
# df.ta.macd(close='close', fast=12, slow=26, signal=9, append=True)

# Get the 26-day EMA of the closing price
k = sprc_df['Close'].ewm(span=12, adjust=False, min_periods=12).mean()

# Get the 12-day EMA of the closing price
d = sprc_df['Close'].ewm(span=26, adjust=False, min_periods=26).mean()

# Subtract the 26-day EMA from the 12-Day EMA to get the MACD
macd = k - d

# Get the 9-Day EMA of the MACD for the Trigger line
macd_s = macd.ewm(span=9, adjust=False, min_periods=9).mean()

# Calculate the difference between the MACD - Trigger for the Convergence/Divergence value
macd_h = macd - macd_s

# Add all of our new values for the MACD to the dataframe
sprc_df['MACD_Con_Di'] = sprc_df.index.map(macd_h)
```

### 3.2.5. Parabolic SAR

We calculate SAR by using the available 'TA-Lib' library in 'talib.SAR' function. An example code is below.

```
# SAR
sprc_df['SAR'] = ta.SAR(sprc_df.High.values, sprc_df.Low.values, acceleration = 0.02, maximum = 0.2)
sprc_df
```

### 3.2.6. SET 50 and SET 100 prices

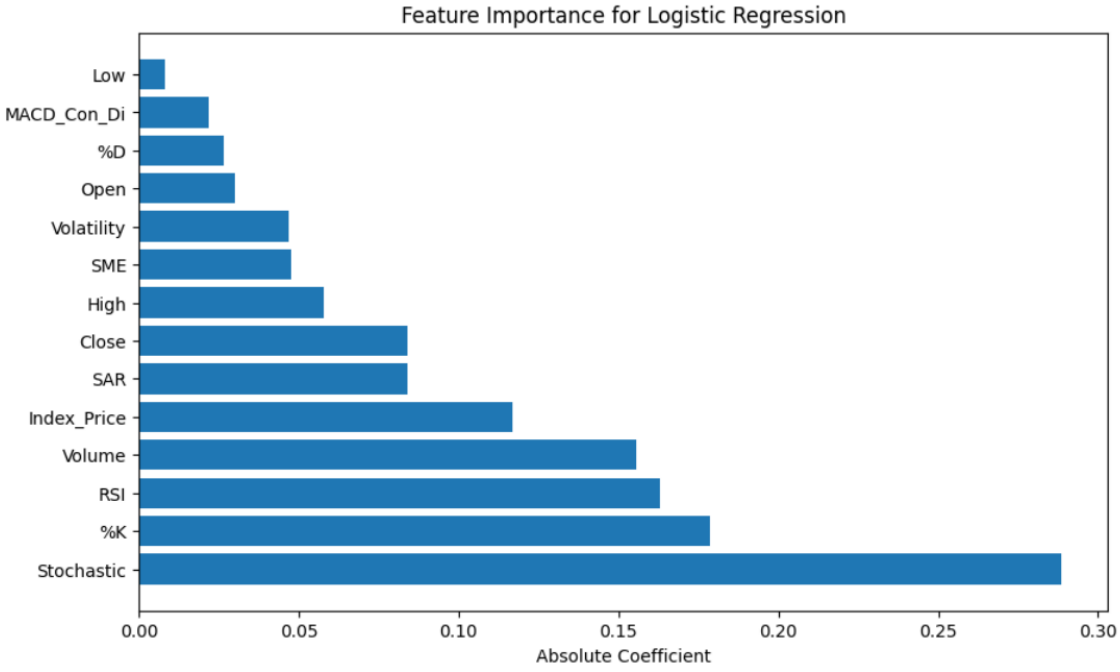
As previously mentioned, we use both price data as features. The SET 50 index price for TISCO model, and The SET 100 index price for ERW and SPRC model. We import historical price data from **Yahoo website** by 'yfinance 'library. We select their 'Close price' as the features.

### 3.2.7. Stochastic Correlation

Firstly, we use only 6 predictors as mentioned to train logistic regression model and analyze the coefficients assigned to each feature. The magnitude of the coefficients gives an indication of the impact of each feature on the predicted outcome. Larger magnitudes suggest a stronger influence. We found that '**Stochastic Oscillator**' was the strongest influence. Therefore, we create relevant a feature, which shows correlation between the **'Close price'** of each stock and '**Stochastic Oscillator**' that can improve the discrimination between classes. The example code to find feature coefficients and rank them.

```
# Get feature coefficients
feature_coefficients2 = best_logis_model2.coef_[0]

# Create a DataFrame to display feature coefficients
feature_importance_df2 = pd.DataFrame({'Feature': X2.columns, 'Coefficient': feature_coefficients2})

# Sort the DataFrame by coefficient magnitude in descending order
feature_importance_df2['Abs_Coefficient'] = np.abs(feature_importance_df2['Coefficient'])
feature_importance_df2 = feature_importance_df2.sort_values(by='Abs_Coefficient', ascending=False)

# Print or visualize the feature importances
print(feature_importance_df2)

# Plotting feature importances
plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df2['Feature'], feature_importance_df2['Abs_Coefficient'])
plt.xlabel('Absolute Coefficient')
plt.title('Feature Importance for Logistic Regression')
plt.show()
```

The result shows that '**Stochastic Oscillator**' was strongest influence.




The code to create 'Stochastic Oscillator' as the following.

```
#Stochastic Correlation
erw_price_df['Sto_Corr'] = erw_price_df['Close'].rolling(window=14).corr(erw_price_df['Stochastic'])
```

## 3.3 EDA and Data preparation

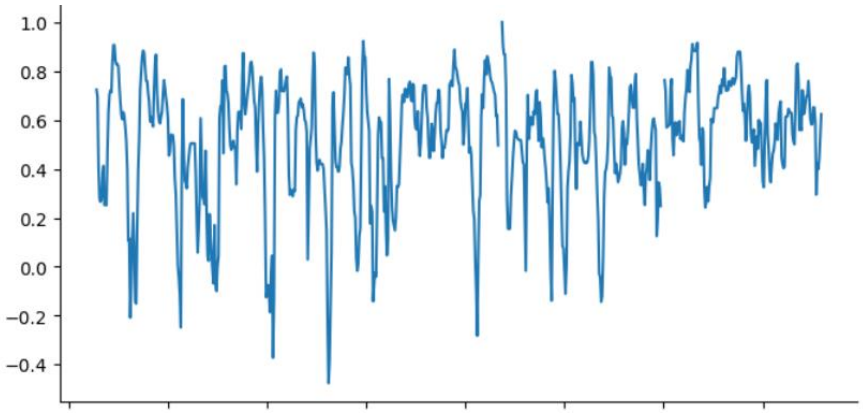
### 3.3.1. Check infinity value

Firstly, we check infinity value and find that 'Stochastic Correlation' contains infinity value.

```python
# Check if there are any infinity values in each column
# Check for infinity values in the DataFrame
is_infinity = np.isinf(erw_price_df)
for column in erw_price_df.columns:
    has_infinity = is_infinity[column].any()
    if has_infinity:
        print(f"Column '{column}' contains infinity values.")
    else:
        print(f"Column '{column}' does not contain infinity values.")
```

```
Column 'Open' does not contain infinity values.
Column 'High' does not contain infinity values.
Column 'Low' does not contain infinity values.
Column 'Close' does not contain infinity values.
Column 'Volume' does not contain infinity values.
Column 'SME' does not contain infinity values.
Column 'RSI' does not contain infinity values.
Column '%K' does not contain infinity values.
Column '%D' does not contain infinity values.
Column 'Stochastic' does not contain infinity values.
Column 'Sto_Corr' contains infinity values.
Column 'Volatility' does not contain infinity values.
Column 'Index_Price' does not contain infinity values.
Column 'MACD_Con_Di' does not contain infinity values.
Column 'SAR' does not contain infinity values.
```

We plot the graph to find the range of value. We find that mostly value is in the range (-1,1).Therefore, we change the positive infinity value to 1 and negative infinity value to -1. The example code is shown below.




*The graph plot 'Stochastic Correlation' value.*

```python
#fill infinity data
def replace_negative_infinity(df, replacement_value):
    # Replace negative infinity with the specified value
    df.replace(-np.inf, replacement_value, inplace=True)
    return df

replacement_value = -1.0  # You can replace this with any value you want

result = replace_negative_infinity(erw_price_df, replacement_value)


def replace_positive_infinity(df, replacement_value):
    # Replace negative infinity with the specified value
    df.replace(np.inf, replacement_value, inplace=True)
    return df

replacement_value = 1.0  # You can replace this with any value you want

result = replace_positive_infinity(erw_price_df, replacement_value)

# Check for infinity values in the DataFrame
is_infinity = np.isinf(result)
for column in result.columns:
    has_infinity = is_infinity[column].any()
    if has_infinity:
        print(f"Column '{column}' contains infinity values.")
    else:
        print(f"Column '{column}' does not contain infinity values.")

erw_price_df = result
erw_price_df
```

```
Column 'Open' does not contain infinity values.
Column 'High' does not contain infinity values.
Column 'Low' does not contain infinity values.
Column 'Close' does not contain infinity values.
Column 'Volume' does not contain infinity values.
Column 'SME' does not contain infinity values.
Column 'RSI' does not contain infinity values.
Column '%K' does not contain infinity values.
Column '%D' does not contain infinity values.
Column 'Stochastic' does not contain infinity values.
Column 'Sto_Corr' does not contain infinity values.
Column 'Volatility' does not contain infinity values.
Column 'Index_Price' does not contain infinity values.
Column 'MACD_Con_Di' does not contain infinity values.
Column 'SAR' does not contain infinity values.
```

### 3.3.2. Check Nan value

Secondly, we check missing data in each column as shown below. The result shows that there are missing values.

```python
def check_nan_data(dataframe):
    nan_counts = dataframe.isna().sum()
    nan_columns = nan_counts[nan_counts > 0]

    if len(nan_columns) == 0:
        print("No NaN values found in any column.")
    else:
        print("NaN values found in the following columns:")
        for column, count in nan_columns.items():
            print(f"{column}: {count} NaN values")

check_nan_data(erw_price_df)
```

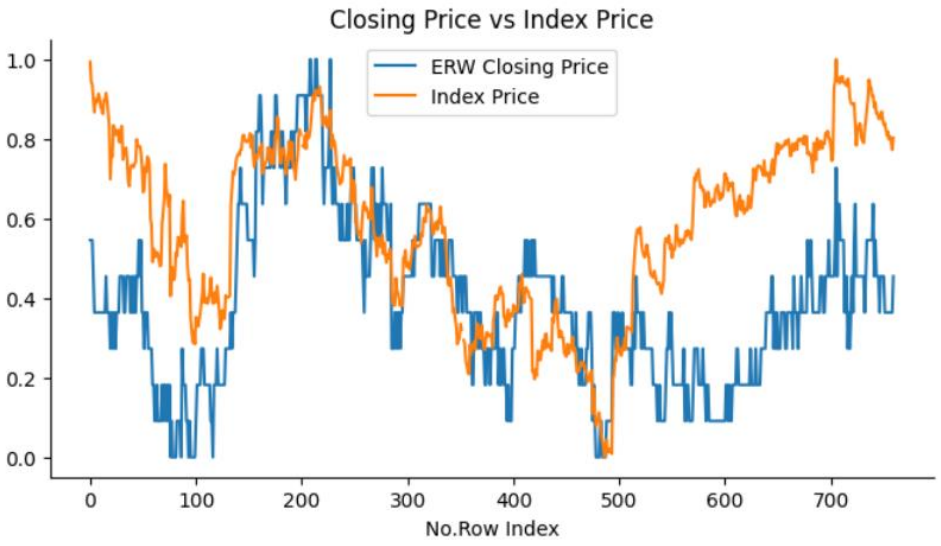
```
NaN values found in the following columns:
SME: 9 NaN values
RSI: 14 NaN values
%K: 13 NaN values
%D: 15 NaN values
Stochastic: 15 NaN values
Sto_Corr: 29 NaN values
Volatility: 5 NaN values
Index_Price: 2 NaN values
MACD_Con_Di: 33 NaN values
SAR: 1 NaN values
```

### 3.3.3. Fill Missing Data

As mentioned in the previous topic, there are columns with missing values. To fill missing values, we use imputation strategy. We will explain how to fill missing values in column 'Index_Price' to give the example.

Firstly, we plot the graph between 'Close price' and 'Index Price' to find the relationship between them. The result shows that 'Close price' and 'Index Price' are in the same trend. This step is to emphasize that 'Close price' is related to the 'Index Price", so keeping the 'Index Price' as predictor may be benefit to our model. To use 'Index Price' as predictor, fill missing value in this column is crucial. The code and graph are shown below.

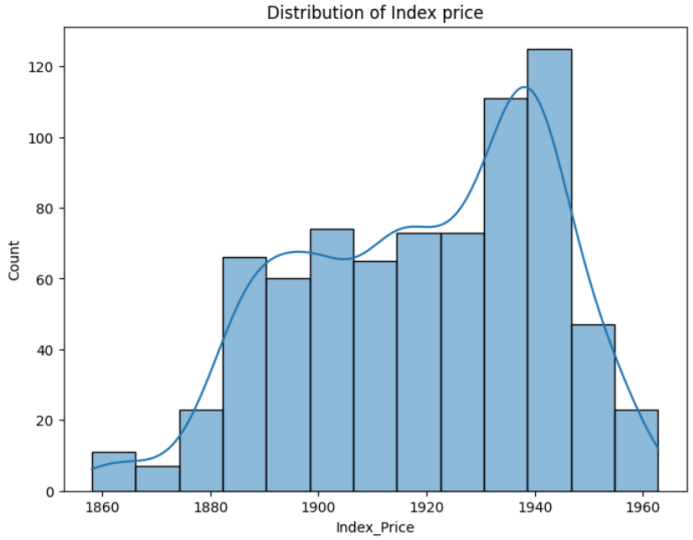
```python
# Function to normalize the DataFrame
def normalize_dataframe(df):
    scaler = MinMaxScaler()
    df_normalized = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
    return df_normalized

# Normalize the DataFrame
normalized_df = normalize_dataframe(erw_price_df)


#Plot Closing price of ERW stock and Index price
# Assuming normalized_df is your DataFrame
ax = normalized_df[['Close', 'Index_Price']].plot(kind='line', figsize=(8, 4), title='Closing Price vs Index Price')
ax.spines[['top', 'right']].set_visible(False)

# Adding x-axis label
plt.xlabel('No.Row Index')

# Adding legends for each line
ax.legend(['ERW Closing Price', 'Index Price'])

plt.show()
```




To fill missing data in 'Index Price', we create a 'SimpleImputer' instance with strategy='mean' to impute the missing data. The reason why we choose 'mean' is value containing missing data is normal distribution. We check the normality by plotting the histogram graph and using Shapiro-Wilk statistic test. An example code to check normality and its result as shown in below figure.

```
#Data distribution for 'Index_Price'

# Check normality using Shapiro-Wilk test
stat, p_value = stats.shapiro(erw_price_df['Index_Price'])
print(f'Shapiro-Wilk test statistic: {stat}, p-value: {p_value}')

# Set significance level
alpha = 0.05

# Check normality based on p-value
if p_value > alpha:
    print("The data follows a normal distribution (Do not reject H0)")
else:
    print("The data does not follow a normal distribution (Do not reject H0)")

# Visualize distribution
plt.figure(figsize=(8, 6))
sns.histplot(erw_price_df['Index_Price'], kde=True)
plt.title("Distribution of Index price")
plt.show()
```

```
Shapiro-Wilk test statistic: nan, p-value: 1.0
The data follows a normal distribution (Do not reject H0)
```




The following code shows how we impute missing value in 'Index Price'.

```
from sklearn.impute import SimpleImputer
# Choose the column with missing data
column_with_missing_data = 'Index_Price'

# Create a SimpleImputer with strategy='mean'
imputer = SimpleImputer(strategy='mean')

# Apply imputation to the specified column
erw_price_df[column_with_missing_data] = imputer.fit_transform(erw_price_df[[column_with_missing_data]])

erw_price_df
```

### 3.3.4. Define Target Variable

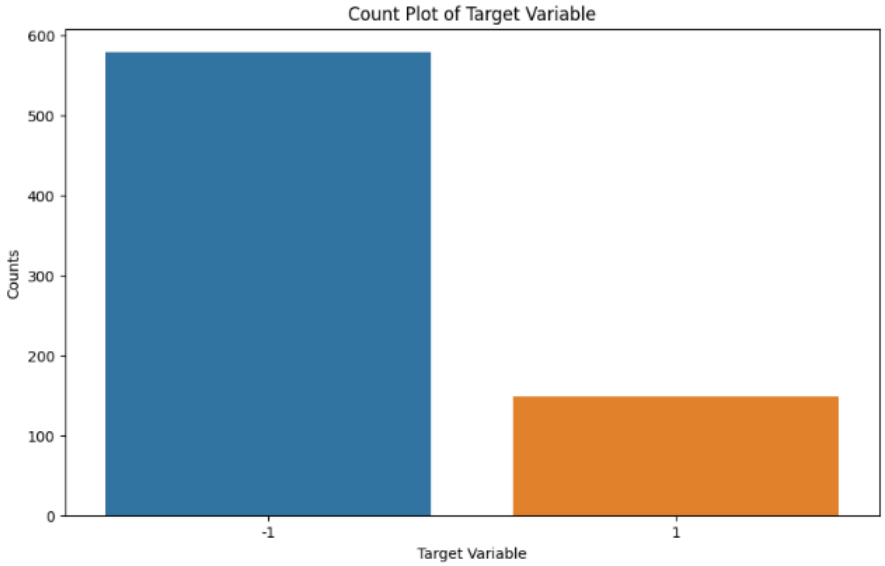
If the closing price in the next 15 minutes is higher than the current closing price, then we will buy the stock (1), else we will sell it (-1)**.**

```
erw_price_df["y"] = np.where(erw_price_df['Close'].shift(-1) > erw_price_df['Close'],1,-1)
erw_price_df
```

### 3.3.5. Explore Imbalance in the target variable

After defining the target variable, we explore the imbalance class which is very common in practical classification scenarios. If we know that there is this problem, any usual approach to solving this kind of problem often yields inappropriate results. We explore it by plot histogram graph to count the number of -1 class and 1 class as show in below.

```
# Use Seaborn to count and plot
plt.figure(figsize=(10, 6))
sns.countplot(x='y', data= erw_price_df)
plt.xlabel('Target Variable')
plt.ylabel('Counts')
plt.title('Count Plot of Target Variable')
plt.show()
```




The result shows that there is an imbalance class in our data. Therefore, solving this problem during training the model should be done to meet satisfying model performance.

## 3.4 Model Training

We train the model by using 5 algorithms including K-Nearest Neighbors, Random Forest, Support Vector Machine, and Logistic Regression. We choose the best model algorithm by comparing the model performance. In this report we will explain only the training process of Logistic Regression algorithm, which is the best classification model algorithm for our classification problem. Let's look at the training process as the following topic.

### 3.4.1 Pre-processing of data

**Split the Dataset:**

We split the dataset into a training dataset and test dataset. We will use 80% of our data to train and the rest 20% to test. To do this, we create a split variable which will divide the data frame in an 80-20 ratio. 'X_train' and 'y_train' are train dataset. 'X_test' and 'y_test' are the test dataset. We set 'stratify' parameter which used to ensure that the class distribution in the target variable is preserved in both the training and testing sets.

```
#Generate Train/Test dataset (80/20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    stratify=y, random_state=123)
```

**Scale the data：**

For logistic regression models, scaling the data before training the model is crucial. We use 'StandardScaler' to standardize the features because our data follows a Gaussian distribution. The example code is shown below.

```
#For X train and X test
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler(with_mean=True, with_std=True)
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Note: We also apply scaling technique to K-NN, and SVM model training.

**Training Process :**

*1. Initialize the model:*

First step, we initialize a Logistic Regression model by defining class_weight, max_iter,random_state, and ,verbose =1.

We adjust class weight to give different importance to different classes in the training process because there is an imbalance in the number of samples across classes as mentioned in 'EDA and data preparation' topic. Due to a lower number of class 1 than class -1, the weight for 1 is higher than -1. The example code is shown in below figure.

```
#Adjust classweight
class_weights = {-1: 1, 1: 2.5}
logis_model = LogisticRegression(class_weight =class_weights,max_iter=500,random_state =123,verbose =1)
```

*2. Cross Validation and hyperparameter tuning:*

We control the generalization of model by finding the right balance between bias and variance. Hyperparameter tuning is our strategy. The focused parameter for tuning includes 'solvers', 'Regularization (penalty)', and 'C parameter' to control the penalty strength. The 'GridSearchCV' was applied to determine the optimal set of hyperparameters. We use 10-fold cross validation to evaluate each model's performance and use `accuracy` as the metrics for model convergence monitoring. After comparing the performance estimates, we choose the hyperparameters settings associated with the best performance. After getting optimal hyperparameters and model selection, we use the best hyperparameter getting from the previous step to fit a model. The example of applying cross validation and hyperparameter tuning as below.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

#Adjust classweight
class_weights = {-1: 1, 1: 2.5}
logis_model = LogisticRegression(class_weight =class_weights,max_iter=500,random_state =123,verbose =1)


solvers = ['liblinear', 'newton-cg', 'lbfgs', 'sag', 'saga']
c_values = np.logspace(-2, 2, 100)
penalty = ['l2','elasticnet']

# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values)
grid_search = GridSearchCV(estimator=logis_model, param_grid=grid, n_jobs=-1, cv=10, scoring='accuracy',error_score=0)
grid_search.fit(X_train_scaled , y_train)
```

## 3.5 Evaluate the model's performance.

After fitting a model, the trained model is ready for prediction. We use `predict_proba` to estimate prediction probability of ' X_test ' set. We use the independent test set to estimate the generalization performance. ROC_AUC score is used to evaluate its performance as shown in the example code.

```
from sklearn.metrics import roc_auc_score
# Get the best estimator from the search
best_logis_model = grid_search.best_estimator_

print("Best parameter")
print(grid_search.best_params_)



# Predict on the test set
logis_y_pred_prob = best_logis_model.predict_proba(X_test_scaled)[:, 1]


# Calculate ROC-AUC score
rf_roc_auc = roc_auc_score(y_test, logis_y_pred_prob)

# Print the ROC-AUC score
print(f"ROC-AUC Score: {rf_roc_auc }")
```

```
Best parameter
{'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
ROC-AUC Score: 0.7566091954022989
```

## 3.6. Train the model with all data and save it for future prediction.

We make use of all our data – merging training and test set– and fit a model to all data points for real-world use.

```
final_logis_model = LogisticRegression(class_weight = {-1: 1, 1: 2.5}, max_iter=500,random_state =123,verbose =1,n_jobs=-1,solver = 'newton-cg',C=0.01,penalty = 'l2')
final_logis_model.fit(X_scaled,y)
final_logis_model
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
LogisticRegression(C=0.01, class_weight={-1: 1, 1: 2.5}, max_iter=500,
                   n_jobs=-1, random_state=123, solver='newton-cg', verbose=1)
```

## 3.7 Algorithm Selection

As mentioned in the above topic, we trained several classification models with the same strategy as topic '3.4 Model Training' including splitting the dataset, scaling the data, and applying cross-

validation and hyperparameter tuning strategy. We will choose the best algorithm by comparing the ROC_AUC score. The result of each model's performance is below table.

| Stock Model | Algorithm | ROC_AUC score |
|---|---|---|
| ERW | K-Nearest Neighbors | 0.572 |
| | Random Forest | 0.705 |
| | Support Vector Machine | 0.675 |
| | Logistic Regression | 0.757 |
| SPRC | K-Nearest Neighbors | 0.637 |
| | Random Forest | 0.708 |
| | Support Vector Machine | 0.708 |
| | Logistic Regression | 0.747 |
| TISCO | K-Nearest Neighbors | 0.699 |
| | Random Forest | 0.759 |
| | Support Vector Machine | 0.784 |
| | Logistic Regression | 0.836 |

The result shows that Logistic Regression algorithms provide the best model performance for all stock models. Therefore, we choose them.

# 4. Conclusions

We trained each classification machine learning model by applying the strategy including train-test data splitting, data scaling, cross-validation and hyperparameter tuning. Then, we evaluated each model algorithm by using 'roc_auc score'. We found that the Logistic Regression achieving the highest roc_auc score,ERW 83.6%, SPRC 74.7% and TISCO 83.6% so Logistic Regression was selected as our solution.