Implementing K-Nearest Neighbor in Trading

Godfred Koduah

Introduction

In the current world, machine learning (ML) has emerged as a disruptive force that has enabled a broad range of applications, from driverless cars to handwriting recognition. Among them, trading has become one area where machine learning has had a significant influence, changing the way strategies are developed and implemented.

With the rise of big data, machine learning has become essential to trading. ML models allow traders to make data-driven decisions by utilizing sentiment research, market indicators, and historical price data. K-Nearest Neighbors (KNN) is a straightforward yet effective supervised learning algorithm that stands out among the many machine learning approaches used in trading.

The K-Nearest Neighbors (KNN) method and its use in trading strategy development are explored in this study. KNN can categorize and forecast trading outcomes by examining past trading data, assisting traders in determining the best times to enter and exit the market.

Why KNN in Trading

KNN is very well-suited for trading due to the following reasons:

1. It operates on the similarity principle, classifying or forecasting data points according to their proximity using a distance metric.

2. For identifying trends in historical data, it is simple yet efficient.

3. By letting consumers try out various choices of k, it balances bias and variation and provides flexibility.

Steps for Implementing KNN in Trading

1. **Data Preparation:** Compile and preprocess historical trading data, making sure it satisfies the input specifications of the KNN algorithm.
2. **Selecting Optimal k:** To determine the ideal ratio of overfitting to underfitting, experiment with different k values.
3. **Defining a Distance Metric:** Utilize metrics like the Manhattan or Euclidean distance to gauge how similar two data points are.
4. **Model Training:** Use previous trading data to train the KNN model, allowing it to pick up trends and behaviors.
5. **Making Predictions:** Using previous similarities, apply the trained model to new market data to forecast trading results.

Objective

This project's main objective is to use KNN to create a trading strategy that can:

- Identify and forecast trade opportunities with accuracy.
- Find trends in previous data to maximize trading performance.
- Use the "collective intelligence" of related data pieces to improve decision-making.

Expected Outcome

Our goal is to have a working KNN-based trading strategy at the project's conclusion, showcasing the algorithm's ability to make data-driven trading decisions. The application will also demonstrate the benefits and drawbacks of applying KNN in a practical trading situation.

Step-by-Step KNN in Python

Import the Libraries

To implement the KNN Algorithm in Python, import the necessary libraries, including numpy for scientific calculations, matplotlib.pyplot for graph plotting, and two machine learning libraries: KNeighborsClassifier and accuracyscore, and the fixyahoo_finance package for Yahoo data fetching.

```
In [1]:
import numpy as np
import pandas as pd
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import RandomizedSearchCV as rcv
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt
from IPython import get_ipython

import matplotlib.pyplot as plt

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

import yfinance as yf

from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Figure 1:

Fetch the data

Now, we will fetch the data using yfinance.

```
SPY_data = yf.download('SPY', start='2014-1-1', end='2024-1-1', auto_adjust=True)
dataframe = SPY_data[['Open', 'High', 'Low', 'Close']]
print(dataframe)
```

```
[*********************100%***********************]  1 of 1 completed

Price             Open        High         Low       Close
Ticker             SPY         SPY         SPY         SPY
Date
2014-01-02  152.119353  152.193777  150.879115  151.242920
2014-01-03  151.499257  151.805191  151.003170  151.218140
2014-01-06  151.714224  151.772095  150.548397  150.779907
2014-01-07  151.383517  151.962292  151.267762  151.705978
2014-01-08  151.681115  151.995312  151.218094  151.738998
...                ...         ...         ...         ...
2023-12-22  469.433280  470.939100  467.293485  469.225250
2023-12-26  469.641316  472.127848  469.562047  471.206543
2023-12-27  470.998521  472.207125  470.453671  472.058533
2023-12-28  472.425074  473.088798  471.810871  472.236847
2023-12-29  472.038690  472.573654  468.878488  470.869720

[2516 rows x 4 columns]
```

Figure 2:

Define Predictor Variable

Predictor variables, like 'Open-Close' and 'High-Low', are used to determine target variable value, with NaN values dropped and stored in 'X' using Python.

```
In [4]:  dataframe['Open-Close'] = dataframe.Open - dataframe.Close
         dataframe['High-Low'] = dataframe.High - dataframe.Low
         dataframe = dataframe.dropna()
         X = dataframe[['Open-Close', 'High-Low']]
         X.head()
```

Out[4]:

Price	Open-Close	High-Low
Ticker		
Date		
2014-01-02	0.876433	1.314662
2014-01-03	0.281118	0.802022
2014-01-06	0.934317	1.223698
2014-01-07	-0.322462	0.694531
2014-01-08	-0.057884	0.777218

Figure 3:

Define Target Variables

The target variable, or dependent variable, is the variable predicted by predictor variables. In this case, the target variable is the SPY price's future closing price. If it exceeds today's price, we will buy SPY, otherwise, we will sell. The buy signal is stored as +1, and the sell signal is stored as -1.

```
Y = np.where(dataframe['Close'].shift(-1) > dataframe['Close'], 1, -1)
```

Figure 4:

Split the Dataset

The dataset will now be divided into training and test sets. 80% of our data will be used for training, with the remaining 20% going toward testing. We will divide the dataframe in an 80-20 ratio by creating a split parameter.

```
In [8]: split_percentage = 0.8
        split = int(split_percentage*len(dataframe))

        X_train = X[:split]
        Y_train = Y[:split]

        X_test = X[split:]
        Y_test = Y[split:]
```

Figure 5:

Instantiate KNN Model

We will instantiate the k-nearest classifier after dividing the dataset into training and test datasets. Since $k = 20$ is being used here, you can adjust the value of k and observe how the outcome changes.

The "fit" function is then used to fit the train data. The "accuracy_score" function will then be used to determine the accuracy of the test and train.

```
In [9]: knn = KNeighborsClassifier(n_neighbors=20)

        knn.fit(X_train, Y_train)

        accuracy_train = accuracy_score(Y_train, knn.predict(X_train))
        accuracy_test = accuracy_score(Y_test, knn.predict(X_test))

        print ('Train_data Accuracy: %.2f' %accuracy_train)
        print ('Test_data Accuracy: %.2f' %accuracy_test)

C:\ProgramData\anaconda3\lib\site-packages\sklearn\neighbors\_classification.py:215: DataConversionWarning: A column-vector
y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using ravel().
  return self._fit(X, y)

Train_data Accuracy: 0.61
Test_data Accuracy: 0.49
```

Figure 6:

Here, we observe a 49% accuracy rate in a test dataset, indicating that our prediction will be accurate 49% of the time.

Create a trading strategy using the model

Buying or selling is our basic trading strategy. Using the predict function, we will forecast if the signal is to buy or sell. Next, we will figure out the test period's total SPY returns.

Next, using the signal that the model predicted in the test dataset, we will compute the cumulative strategy return.

Next, we will visualize the trading strategy's performance using the KNN Algorithm by plotting the cumulative returns of the SPY and the strategy.

In [22]:

```
dataframe['Predicted_Signal'] = knn.predict(X)

dataframe['SPY_data_returns'] = np.log(dataframe['Close']/dataframe['Close'].shift(1))
Cumulative_SPY_data_returns = dataframe[split:]['SPY_data_returns'].cumsum()*100

dataframe['Strategy_returns'] = dataframe['SPY_data_returns']* dataframe['Predicted_Signal'].shift(1)
Cumulative_Strategy_returns = dataframe[split:]['Strategy_returns'].cumsum()*100


plt.figure(figsize=(10,5))
plt.plot(Cumulative_SPY_data_returns, color='r',label = 'SPY Returns')
plt.plot(Cumulative_Strategy_returns, color='g', label = 'Strategy Returns')
plt.legend()
plt.show()
```

Figure 7:

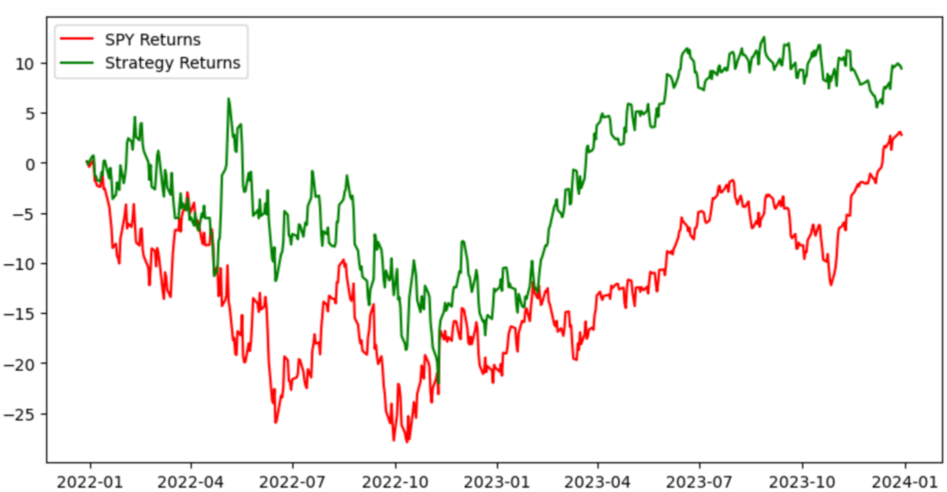

Figure 8:

The cumulative returns of the SPY index and the trading strategy based on the K-Nearest Neighbors (KNN) classifier's expected signals are shown in the figure above.

In summary, the figure contrasts the cumulative returns of the trading strategy (shown by the green line) with the performance of the SPY index (shown by the red line).

When compared to keeping the SPY stock with no trading activity, it enables us to evaluate how well the trading strategy generates returns.

Sharpe Ratio

The return obtained above the market return per unit of volatility is known as the Sharpe ratio. In order to determine the Sharpe ratio, we will first compute the standard deviation of the cumulative returns.

```
In [23]:  Std = Cumulative_Strategy_returns.std()
          Sharpe_ratio = (Cumulative_Strategy_returns - Cumulative_SPY_data_returns)/Std
          Sharpe_ratio = Sharpe_ratio.mean()
          print('Sharpe ratio: %.2f'%Sharpe_ratio )

          Sharpe ratio: 1.24
```

Figure 9:

With a Sharpe ratio of 1.24, the strategy or investment has produced a return that is 1.24 times higher than the amount of risk assumed per unit.

Generally speaking, a Sharpe ratio greater than 1 is regarded as favorable. To better comprehend the Sharpe ratio's relative performance, it's crucial to compare it to other investment options or benchmarks.