K-Nearest Neighbors

Kannan Singaravelu, CQF

June 2023

1 K-Nearest Neighbors

The K-Nearest Neighbors (KNN) is one of the simplest machine learning algorithms that can be used for both regression and classification. KNN is a lazy learning algorithm and doesn't technically train a model to make predictions. Instead an observation is predicted to be the class of that of the largest proportion of the k nearest observations. KNN method is a direct attempt at approximating the conditional expectation using actual data.

For **regression**, the predicted value is the mean of the K neighbors and the estimator is given as

$$\hat{f}(x) = Average\left[y_i | x_i \in \mathcal{N}_k(x)\right]$$

where, $\mathcal{N}_k(x)$ is a neighborhood of x containing the k closest observations.

For **classification**, the predicted label is the class with the plurality, i.e., which class is most represented among the neighbors. This is equivlent to taking a majority vote among the k nearest neighbors. For each class $j = 1, \cdots, K$, we then compute the empirical (conditional) probability

$$Pr(G = j | X = x_0) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x)} I(y_i = j)$$

and assign the observation to the class with the highest empirical probability. Here, $I(y_i = j)$ is the indicator function returning 1 if $y_i = j$ and 0 otherwise.

1.1 The Three Steps

The popularity of KNN comes from the fact that it is very easy to understand and interpret. It's accuracy is often comparable or even better than more complicated algorithms. Once, k is specified, finding nearest neighbors is a 3-step process.

Steps	Remarks
Step 1	Compute the distance, usually Euclidian
Step 2	Sort by ascending distance to find the k NN
Step 3	Compute the average or probability of the k NN observations

1.2 Finding the neighbors

Intuitively, distance can be thought of as a measure of similarity. Euclidean distance is the most commonly used but other distance metrics such as Manhattan work as well. The generalized distance metric is called the Minkowski distance, defined as

$$d = \left(\sum_{n=i}^{n} |x_i - y_i|^p \right)^{1/p},$$

where x_i and y_i are the two observations for which distance d is being calculated with a hyperparameter, integer p.

When $p = 1$, the Minkowski distance is the Manhattan distance and when $p = 2$, the Minkowski distance is the just the standard Euclidean distance. With the K neighbors identified using distance metrics, the algorithm can make a classification or prediction with the label values of the neighbors.

1.3 Selecting K

KNN is a non-parametric method and it does not assume any function as there are no parameters to be estimated. The selection of the number of neighbors k is done using the training set. Selecting k close to 1 gives the most flexibility (low bias), but also has the highest variability (high variance) while on the other hand, selecting a large k gives the least flexibility (high bias), but also has the lowest variability (low variance). In particular, $k = N$ will assign all new test observations to a single class. The best way of selecting k is through cross validation. Another alternative approach is to employ the popular Elbow method to select k.

2 Problem Statement

The objective is to predict market movement based on classification algorithm. In this lab, we'll use KNN to predict market direction and devise a trading strategies based on it.

Import Libraries

[ ]:
```
# Base Libraries
import pandas as pd
import numpy as np

# Plotting
import matplotlib.pyplot as plt
# plt.rcParams['figure.figsize'] = (20,7)
# plt.style.use('fivethirtyeight')

# Preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import (
                                    train_test_split,
                                    GridSearchCV,
```

```
                                    TimeSeriesSplit,
                                    cross_val_score
                                    )
# Classifier
from sklearn.neighbors import KNeighborsClassifier

# metrics
from sklearn.metrics import (precision_recall_curve,
                             roc_curve,
                             RocCurveDisplay,
                             ConfusionMatrixDisplay
                            )

from sklearn.metrics import (accuracy_score,
                             f1_score,
                             recall_score,
                             precision_score,
                             roc_auc_score,
                             auc
                            )

from sklearn.metrics import (classification_report,
                             confusion_matrix
                            )

# Ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

2.1 Load Data

[ ]:
```
# Load locally stored data
df = pd.read_csv('data/spy.csv', index_col=0, parse_dates=True)

# Check first 5 values
df.head()
```

[ ]:
```
df.shape
```

[ ]:
```
# Visualize data
plt.plot(df['Adj Close']);
```

2.2 EDA of Original dataset

[ ]:
```
# Descriptive statistics
df.describe()
```

2.3 Cleaning & Imputation

Data is already cleaned. No further processing or imputation required.

```
[ ]: # Check for missing values
df.isnull().sum()
```

2.4 Feature Specification

Features or Predictors are also known as an independent variable which are used to determine the value of the target variable. We will derive a features set from the original dataset.

```
[ ]: # Predictors
df['O-C'] = df.Open - df.Close
df['H-L'] = df.High - df.Low

X = df[['O-C', 'H-L']].values
X
```

2.5 Target or Label Definition

Label or the target variable is also known as the dependent variable. Here, the target variable is whether Nifty Index price will close up or down on the next trading day. If the tomorrow's closing price is greater than the 0.995 of today's closing price, then we will buy the Nifty, else we will sell the index.

We assign a value of +1 for the buy signal and -1 for the sell signal to target variable. The target can be described as :

$$y_t = \begin{cases} +1, & \text{if } p_{t+1} > 0.995 * p_t \\ -1, & \text{if } p_{t+1}\text{Otherwise} \end{cases}$$

whre, p_t is the current closing price of Nifty Index and p_{t+1} is the 1-day forward closing price of the index.

```
[ ]: # Target- Avoid using [-1, 1], always prefer [0, 1] as class labels
y = np.where(df['Adj Close'].shift(-1)>0.995*df['Adj Close'],1,-1)
y
```

```
[ ]: # Value counts for class 1 and -1
pd.Series(y).value_counts()
```

Split Data

```
[ ]: # Splitting the datasets into training and testing data.
# Always keep shuffle = False for financial time series
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪shuffle=False)
```

```
# Output the train and test data size
print(f"Train and Test Size {len(X_train)}, {len(X_test)}")
```

3 Base Model

We now build a base model with default parameters using Pipelines. Since the KNN model calculates distances, the dataset needs to be scaled for the model to work properly. All the features should have a similar scale. The scaling can be accomplished by using the `MinMaxScaler` transformer.

3.1 Fit Model

[ ]:
```
# Scale and fit the model
model = Pipeline([
    ("scaler", MinMaxScaler()),
    ("classifier", KNeighborsClassifier())
])

model.fit(X_train, y_train)
```

3.2 Predict Model

[ ]:
```
# Predicting the test dataset
y_pred = model.predict(X_test)

# Predict Probabilities
y_proba = model.predict_proba(X_test)
```

[ ]:
```
# verify the class labels
model.classes_
```

[ ]:
```
# predict probability
y_proba[-20:]
```

[ ]:
```
# predict class labels
y_pred[-20:]
```

[ ]:
```
acc_train = accuracy_score(y_train, model.predict(X_train))
acc_test = accuracy_score(y_test, y_pred)

print(f'Train Accuracy: {acc_train:0.4}, Test Accuracy: {acc_test:0.4}')
```

3.3 Prediction Quality

3.3.1 Confusion Matrix

Confusion matrix is a table used to describe the performance of a classification model on a set of test data for which the true values are known.

Outcome	Position[1]
True Negative	upper-left
False Negative	lower-left
False Positive	upper-right
True Positive	lower-right

True Positive is an outcome where the model correctly predicts the positive class. Similarly, a true negative is an outcome where the model correctly predicts the negative class.

False Positive is an outcome where the model incorrectly predicts the positive class. And a false negative is an outcome where the model incorrectly predicts the negative class.

Note: In a binary classification task, the terms ''positive" and ''negative" refer to the classifier's prediction, and the terms ''true" and ''false" refer to whether that prediction corresponds to the external judgment (sometimes known as the ''observation") and the axes can be flipped. Refer Scikit-Learn Binary Classification for further details.

[ ]:
```
# Display confussion matrix
disp = ConfusionMatrixDisplay.from_estimator(
        model,
        X_test,
        y_test,
        # display_labels=model.classes_,
        cmap=plt.cm.Blues
    )
plt.title('Confusion matrix')
plt.show()
```

3.3.2 Classification Report

A classification report is used to measure the quality of predictions from a classification algorithm.

[ ]:
```
# Classification Report
print(classification_report(y_test, y_pred))
```

Macro Average Average of precision (or recall or f1-score) of different classes.

Weighted Average Actual Class1 instance * precision (or recall or f1-score) of Class1 + Actual Class2 instance * (or recall or f1-score) of Class2.

3.3.3 Receiver Operator Characteristic Curve (ROC)

The area under the ROC curve (AUC) is a measure of how well a model can distinguish between two classes. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various classification thresholds.

[ ]:
```
# Display ROCCurve
disp = RocCurveDisplay.from_estimator(
            model,
```

```
            X_test,
            y_test,
            name='Tuned KNN')
plt.title("AUC-ROC Curve \n")
plt.plot([0,1],[0,1],linestyle="--", label='Random 50:50')
plt.legend()
plt.show()
```

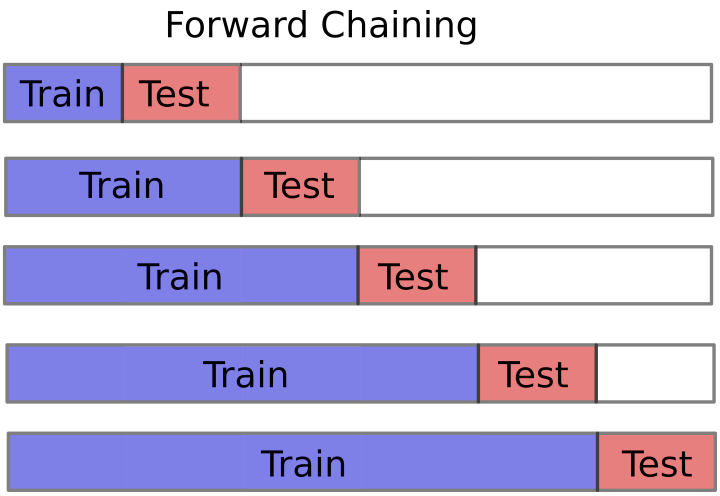
4 Hyper-parameter Tuning

Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. It is possible and recommended to search the hyper-parameter space for the best cross validation score. Any parameter provided when constructing an estimator may be optimized in this manner.

4.1 Cross-validation of Time Series

Time series data are sequential in nature and are characterised by the correlation between observations. Classical cross-validation techniques such as KFold assume the samples are independent and identically distributed, and would result in poor estimates when applied on time series data.

To preserve the order and have training set occur prior to the test set, we use **Forward Chaining** method in which the model is initially trained and tested with the same windows size. And, for each subsequent fold, the training window increases in size, encompassing both the previous training data and test data. The new test window once again follows the training window but stays the same length.



We will tune the hyperparameters to select the K-Best Neighbor by **TimeSeriesSplit** from scikit-learn. This is a forward chaining cross-validation method and is a variation from the KFold. In the kth split, it returns first k folds as train set and the (k+1)th fold as test set. Unlike standard cross-validation methods, successive training sets are supersets of those that come before them.

[ ]:
```
# Example: First 2 split
tscv = TimeSeriesSplit(n_splits=4, gap=1)
for train, test in tscv.split(X):
    print(train, test)
```

[ ]:
```
# Cross-validation
tscv = TimeSeriesSplit(n_splits=5, gap=1)
```

4.2 GridSearch

The conventional way of performing hyperparameter optimization has been a grid search (aka parameter sweep). It is an exhaustive search through a manually specified subset of the hyperparameter space of a learning algorithm. A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set or evaluation on a validation set.

GridSearch performs exhaustive search over specified parameter values for an estimator. It implements a "fit" and a "score" method among other methods. The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

[ ]:
```
# Get parameters list
model.get_params()
```

[ ]:
```
# Perform Gridsearch and fit
param_grid = {"classifier__n_neighbors": np.arange(1,51,1)}

gs = GridSearchCV(model, param_grid, scoring='roc_auc', n_jobs=-1, cv=tscv,␣
 ↪verbose=1)
gs.fit(X_train, y_train)
```

[ ]:
```
# Best Params & Scores
print(f"Optimal Neighbours: {gs.best_params_['classifier__n_neighbors']}, Best␣
 ↪Score: {round(gs.best_score_,4)}")
```

5 Tuned Model

Let's now train and predict the model with the best search parameter

[ ]:
```
# Instantiate KNN model with search param
clf = KNeighborsClassifier(n_neighbors = gs.
 ↪best_params_['classifier__n_neighbors'])

# Fit the model
clf.fit(X_train, y_train)
```

[ ]:
```
# Predicting the test dataset
y_pred = clf.predict(X_test)

# Predict Probabilities
```

```
# y_proba = clf.predict_proba(X_test)[:,1]

# Measure Accuracy
acc_train = accuracy_score(y_train, clf.predict(X_train))
acc_test = accuracy_score(y_test, y_pred)

# Print Accuracy
print(f'\n Training Accuracy \t: {acc_train :0.4} \n Test Accuracy \t\t:
 ↪{acc_test :0.4}')
```

[ ]:
```
# Display confussion matrix
disp = ConfusionMatrixDisplay.from_estimator(
        clf,
        X_test,
        y_test,
        display_labels=model.classes_,
        cmap=plt.cm.Blues
    )
plt.title('Confusion matrix')
plt.show()
```

[ ]:
```
# Display ROCCurve
disp = RocCurveDisplay.from_estimator(
            clf,
            X_test,
            y_test,
            name='Tuned KNN')
plt.title("AUC-ROC Curve \n")
plt.plot([0,1],[0,1],linestyle="--", label='Random 50:50')
plt.legend()
plt.show()
```

[ ]:
```
# Classification Report
print(classification_report(y_test, y_pred))
```

Observation

1. Test accuracy improved more than 5% as compared to the earlier model.
2. Recall for Class +1 improved by more than 7% as compared to the earlier model while recall for Class -1 decreased by 40%.
3. Model improved predictor for upside when compared to the downside.
4. Class imbalance skews the prediction and needs to be addressed.

6 Trading Strategy

Let's now define a trading strategy. We will use the predicted signal to buy or sell. We then compare the result of this strategy with the buy and hold and visualize the performance of the KNN Algorithm.

[ ]:
```
# Subsume into a new dataframe
df1 = df.copy()                          # df[-len(X_test)
df1['Signal'] = clf.predict(X)           # clf.predict(X_test)
```

[ ]:
```
# Daily Returns - Benchmark return
df1['Returns'] = np.log(df1['Adj Close']).diff().fillna(0)

# Strategy Returns - KNN
df1['Strategy'] = df1['Returns'] * df1['Signal'].shift(1).fillna(0)

# Localize index for pyfolio
df1.index = df1.index.tz_localize('utc')
```

[ ]:
```
# Check the output
df1.tail(10)
```

6.1 Return Analysis

[ ]:
```
# Import pyfolio
import pyfolio as pf
```

[ ]:
```
# Create Tear sheet using pyfolio for outsample - for X_test
# pf.create_simple_tear_sheet(df1['Strategy'])
```

[ ]:
```
# Live start date 2016-04-07
pf.create_returns_tear_sheet(df1['Strategy'], live_start_date='2016-04-07',␣
 ↪benchmark_rets=df1['Returns'])
```

7 References

- TimeSeriesSplit
- Cross-validation
- GridSearchCV
- Hyperparameters Tuning
- K-Neighbors Classifier

Notes 1. Sckit-learn format. One may also use a difference convention for axes.

June 2023, Certificate in Quantitative Finance.