Documentation of FXVOLAUTOMAT v2.0.py

The following code is an extension of FXVOLAUTOMAT v1.0.py. We use the functions in 1.0.py to read and pre-process the dataset and extend these to apply ML techniques like CART, SVM, Logistic Regression, Ridge Classifiers and Random Forests to run a predictive 2-class classification problem. The 2 classes being 1: go long the 1M EURUSD ATM VOL and 0: Go short the same. We tune the Hyper-parameters for each of the ML techniques using Time-Series Cross-validation which is essentially an expanding window scheme and record the Out of Sample accuracies of each algorithm.

Input Datasets (in same directory as code)

1. filename - **"DataTables.xlsx"** (Daily price data for Predictors dataset)

2. filename – "**data_util.py**" (separate script that contains standard utility functions. In particular, the following 3 functions are in data_util and called by v2.0.

   def printingStatistics(df)
   def maximumDrawdown(ret_series)
   def monthly_performance_stats(weightdRet)

   Note in order to call these functions from the v2.0 script, we import the data_util script as if it were a package using the command **import data_util as du**

The functions used in this script are:

def optimizeHyperParameters(train_X,train_Y,test_X,test_Y, parameter, algo, cv_folds)

def plot_parameters_vs_accuracies(parameters, test_accuracy, train_accuracy,x_label,title)

def evaluate_performance(model,pred,train_X,train_Y,test_X,test_Y)

def knnmodel(train_X,train_Y,test_X,test_Y)

def plotROC(test_Y,y_pred_prob)

def logisticregression(train_X,train_Y,test_X,test_Y)

def ridge_Classification(train_X,train_Y,test_X,test_Y)

def Linear_SVM(train_X,train_Y,test_X,test_Y)

def CART(train_X,train_Y,test_X,test_Y)

def RandomForest(train_X,train_Y,test_X,test_Y)

def euro_vanilla(S, K, T, rd,rf, sigma, option = 'call')

def getData()

def getDependentVariable(method = "BS")

def traintestsplit(X,Y,split)

def standardize(trainX,testX)

def plot_pca_components(x,y)

def model_pca(variance)

The In-sample and Out-of-sample accuracies for all the ML algorithms is as follows:

Training Accuracies	Raw Data	Normalised	PCA 98.6	PCA 91.9	PCA 86.1
KNN	100.00%	92.72%	92.58%	92.58%	92.86%
Logistic	79.79%	86.97%	100.00%	100.00%	70.02%
Ridge	86.78%	86.78%	76.92%	66.43%	64.32%
CART	89.80%	89.80%	74.62%	63.75%	63.60%
SVM	57.76%	60.78%	60.63%	60.30%	59.72%
Random Forest	91.14%	79.26%	59.82%	59.82%	60.39%

Test Accuracies	Raw Data	Normalised	PCA 98.6	PCA 91.9	PCA 86.1
KNN	52.11%	58.24%	56.13%	55.75%	56.70%
Logistic	52.11%	49.62%	48.66%	48.28%	48.47%
Ridge	43.49%	43.49%	48.28%	54.79%	51.15%
CART	47.70%	46.74%	48.85%	59.39%	58.05%
SVM	53.26%	49.62%	50.38%	49.81%	49.43%
Random Forest	55.36%	59.77%	54.79%	54.21%	53.45%

The confusion Matrix and the Classification reports for the Normalised dataset run on kNN is as

Confusion Matrix - kNN Normalised (OOS Data)		Predicted	
		Short	Long
Actual	Short	222	100
	Long	118	82

Classification Report				
	Precision	Recall	f1-score	support
0	67%	44%	53%	322
1	42%	65%	51%	200
avg/total	57%	52%	52%	522

The structure of the code is as follows: POINTS (1-6 are the same ones form version 1.0 to maintain continuity)

1. Read in the predictors from the **"DataTables.xlsx"** using function **getData().** The aim is to try and model the predictors based on the JPM FX Vol Automat research paper. The predictor list as defined by the paper is shown as follows for reference:

Table 1: Our dataset consists of 377 indicators from FX and cross-asset markets as well as macro data

A total of 377 market indicators are formed across various asset classes. We use a 10Y history of these indicators, with daily sampling.

Market data type	Market data	Level	1 week change	1M change	Count
FX realised vols	2M realised vols in USD vs G10, MXN, BRL, ZAR, TRY, INR and KRW	X	X	X	45
FX ATM vols	1M, 3M and 1Y ATM in same pairs	X	X	X	135
FX skews	3M 25D RRs	X	X	X	45
FX spots	FX Spots in 15 G10 and EM USD pairs		X	X	30
Depo rates / Basis	3M FX Forward drops		X	X	30
Interest Rates	10Y Gov yields: US, Japan, UK, Germany, France, Italy, Spain, Australia		X	X	16
Equity Indices	S&P, Nikkei, FTSE, E-Stoxx, ASX, Mexbol, Bovespa, KOSPI, Hang Seng		X	X	18
Commodities	Gold and Brent spot		X	X	4
Credit spreads	CDX IG and HY, iTraxx spread indices		X	X	6
EASI indices	Global, US, CAD, EU, UK, CHF, NOK, SEK, Japan, AU, NZ, China		X	X	24
IMM positions	USD, EUR, JPY, GBP, CHF, CAD, AUD, NZD, MXN, RUB, Gold		X	X	24

Source: J.P. Morgan.

a. The predictor data is stored in the above xlsx file under the following sheet names:
    i. FX SPOT, ATM VOLS, 3M 25D RR, 3M DEPOSIT RATES, 10Y YIELD, EQUITY INDICES, COMDTY, CREDIT SPREADS, JPM EASI, IMM POSITIONING
b. We store the daily levels for the following predictors: FX 2M REALIZED VOLS, FX ATM VOLS, and FX SKEWS
c. For all the remaining predictors we also alculate both the 1 week change and 1M change and store these.
d. The dataframe **df_main** stores all the modified 373 predictors to be further used for data manipulation and input to ML algorithms

2. We use 2 methods to calculate the dependent variable namely – Black Scholes Calculation method and the Realized – Implied Volatility measure. We use the function **getDependentVariable(method = "BS")** to pre-process the dependent variable and store this in the dataframe **Y**
    a. When method is “BS” -
        i. We calculate the value of the ATM Put and ATM Call Options on the daily dataset using the Black Scholes option pricing formula defined for FX options via function **euro_vanilla(S, K, T, rd,rf, sigma, option = 'call')**
        ii. The call and put option prices are added to get the price of the ATM Straddle and stored in dataframe **option**
        iii. We calculate % gain in the price from the 1M ahead spot to current Strike price to calculate in the moneyness of the ATM straddle, subtract this from the cost of the ATM straddle from the above step and store this in **option["Payoff"]**
        iv. We segregate this data set to find a uniformly distributed percentages of positive payoffs and negative payoffs and store these as +1 and 0 respectively in dataframe **Y**
    b. When method is “RV- IV” –
        i. We calculate the 1M ahead realized volatility for EURUSD using the daily returns for EURUSD spot stored in **df_returns["EURUSD CURNCY"]** and store this in dataframe - **df_1MRV**

ii. **RV_IV** dataframe then holds the realized – Implied vols difference
iii. We segregate this data set to find a uniformly distributed percentages of positive payoffs and negative payoffs and store these as +1 and -1 respectively in dataframe **Y**

3. Once the dependent variable stored in Y and predictors stored in **df_main** are calculated, we resample both these to Business Month data sample only from period 2007 to 2016 as done in the JPM paper.
4. We use an 80% and 20% split ratio for the training and test data set and split these using the function **traintestsplit(X,Y,split)**
5. We standardize the predictors as the next step in data processing using the function **standardize(trainX,testX)**
6. The next step towards data processing is to get the reduced dimensionality predictors/features using Principal Component Analysis for 86.1%, 91.9% and 98.6% explained variance of the dataset using the function **runPCA(train_X_stdz,test_X_stdz)**
7. We initialise 2 dataframes **training_results** and **test_results** which will hold the training and the out of sample test accuracies for all our models.
8. Each of the models are implemented in functions named in the function repository above and require only 4 inputs: training samples for X and Y and the test samples for X and Y which are passed depending upon whether we want the models to run on PCA components or raw or normalised data.
9. The general architecture for all the ML algorithms remains the same. For each algorithm we define the hyper-parameter which regularizes the algorithms to avoid overfitting to the training dataset.
10. Hyper-Parameters for each of the algorithms is defined below:
    a. kNN – number of neighbours
    b. logistic regression – degree of polynomial features used on the predictors
    c. Ridge Classifier – L2 parameter
    d. Linear SVM – margin width
    e. Decision Trees/CART – depth of the tree
    f. Random Forests – depth of each tree, rest of the random forest parameters are kept static for simplicity
11. We go in-depth for kNN model in this documentation.
    a. We convert the training and test dataframes into numpy arrays for processing.
    b. We create a list neighbours for the hyper-parameter for this model and define a set of values for the same. For kNN we would like to fit/test the model on 1- 10 nearest neighbours.
    c. We pass this list of hyper-parameters and the algorithm name – 'kNN' to the function **optimizeHyperParameters()** which uses Time-Series Cross validation and returns the best hyper-parameter for which the model returns the highest Cross Validation set accuracy.
        i. Time-Series Cross validation for 5 folds further divides the training set in 5 TRAIN and 5 CV sets as shown below:

```
1. FOLD 1: TRAIN: [0] CV: [1]
2. FOLD 2: TRAIN: [0 1] CV: [2]
3. FOLD 3: TRAIN: [0 1 2] CV: [3]
4. FOLD 4: TRAIN: [0 1 2 3] CV: [4]
5. FOLD 5: TRAIN: [0 1 2 3 4] CV: [5]
```

ii. For every item in the list of hyper parameters, we fit the model in the TRAIN set and calculate the models accuracy on the CV set.
iii. We then average the CV accuracy across all the 5 folds for each hyper-parameter
iv. We compare the average CV accuracies across all the hyper-parameters and find the index of the hyper-parameters which gives the highest accuracy on the cross-validation set
v. The index is then passed on as a result of the **optimizeHyperParameters()** function

d. Once the hyper-parameter is identified for kNN/model we then fit the model to the entire training set using function **model.fit()**
e. We predict the dependent variable Y on the test set by passing it to the **model.predict()** function
f. We print the model performance on the test set using the function **evaluate_performance()** which generates and prints the confusion matrices and the Classification reports for the model.

12. The test set predictions from the kNN model for the Normalised dataset are used further to generate the trading signals based on which we can generate a simple trading strategy.
    a. Predictions 1: go long EURUSD 1M ATM : signal +1
    b. Predictions 0: go short EURUSD 1M ATM : signal -1
    c. The above signals are then stored in the dataframe **df_Signal** and rebalanced monthly to get the monthly signals in dataframe **df_Signal_Monthly**
13. We the find out the strategy returns by multiplying the signal dataframe **df_Signal_Monthly** with the returns stored in **option["Payoff"]** from step 2.a.iii and calculate performance statistics as done for other trading strategies.