Proof of Concept Implementation of: A Rule-based Decision Making system for Automated Risk Management

How to Run the Project

Before running the project, the following commands need to be run to ensure the correct modules are installed and up-to-date. This is as the program requires up to date versions of these modules.

```
pip install --upgrade pandas
pip install arch
pip install yahooquery
```

Once done, the project can be ran by simply running:

```
python3 main.py
```

There are certain customisations possible. Namely, the variable `ticker` (at the top of the main file) can be set to a US ticker, and the output rule set will change based on the ticker provided.

Additionally, the output outcomes (that is, what our rules aim to target) can be specified in the line, in `output=[]` in the line:

. The possible specifications are `['relVaR', 'absVaR', 'relVaREWMA', 'absVaREWMA', 'relVaRGARCH', 'absVaRGARCH', 'CAGR', 'volatility', 'sharpe', 'calmar']`

If no specifications are provided, then the entire list is used.

Lastly, an arbitrary list of stocks can be fed into the datatable creation by altering `x = Datatable(list_of_stocks)`. If no list is provided, the program will instead import an already calculated set of data from the file `DATASET.csv`. This is as importing and pre-calculating the data takes an incredibly long time - for the list of Russell 3000 stocks, it takes around 40 minutes to an hour.

Code Explanation

Importing Data & Creating the Datatable

First, the project imports a large variety of data from a variety of sources - largely Yahoo Finance via. `yahooquery`, but the program also scrapes dividend data from the NASDAQ API because `yahooquery` fails to provide consistent clean data regarding dividends. The program uses a variety of modules from `yahooquery`, namely `financial_data`, `key_stats`, `p_company_360`, and some of `summary_detail`. This data is used for future rules training.

This can be run on any set of US-listed stocks, but for training purposes, we use a csv containing all stocks in the Russell 3000, which is a collection of the ~3000 largest US stocks by market capitalisation. This list is contained within the project files as `Russell3000.csv`.

Note that the importing of data from `Ticker.p_company_360` requires a premium Yahoo Finance subscription. During the development of this project, a trial account was used.

Additionally, the program calculates a variety of metrics using data associated with each stock. These are as follows:

Value at Risk calculations (1 day):

- **relVaR**, **absVaR**: relative and absolute calculations using the standard Value At Risk formula
- **relVaREWMA**, **absVaREWMA**: relative and absolute calculations using an exponentially weighted moving average to calculate volatility. This means that more recent data is weighted exponentially more than older data.
- **relVaRGARCH**, **absVaRGARCH**: relative and absolute calculations using a GARCH model to estimate the Value at Risk. This model is an adaptation of the model in Jon Danielsson's *"Financial Risk Forecasting"* (2011).

Others:

- **CAGR**: the Cumulative Annual Growth Rate for the stock

- **SharpeRatio**: the Sharpe Ratio for the stock, which is a measure of portfolio risk. The assumption is that the associated portfolio is simply a portfolio with 1 stock, rather than a varied portfolio.
- **CalmarRatio**: the Calmar Ratio for the stock, an alternative measure for portfolio risk - which also uses the same assumption as for the Sharpe Ratio.
- **Volatility**: the yearly volatility for the stock.

Pruning Features

Once this is all calculated, we then move onto pruning and normalisation of our data.

Our first function, `PruneFeatures()`, performs a variety of pruning operations. It drops rows from data based on the value of the column NoData (which contains stocks with NaNs above the arbitrary threshold `len(financial_keys)`, which is around 30 NaNs, due to the lack of utility they would have in predicting many stocks. This reflects the human psychological tendency to be conservative when making judgements with little information. It drops `dividends_all`, and various other irrelevant data (such as hiring). It then calls another function, `DataProcess.removeFeatures()`, to combine highly correlated features in data and reduce the total number of features. Once this is processed, It calculates the percentage of NaN values in each remaining feature and prints the results. Finally, it returns the processed data DataFrame

`DataProcess.removeFeatures()` calculates the correlation matrix corrmatrix between the features in data specified by xcol, using the `DataProcess.crosscorr()` function, which is an implementation of the cross-correlation matrix that allows for different types of cross-correlation (e.g. ‘pearson’, ‘kendall’, etc.). This custom implementation is necessary as default implementations crash when dealing with NaN values, which is not a desired characteristic. It then drops any columns and rows that are all NaN values in corrmatrix. It applies the `DataProcess.linearfeatures()` function to corrmatrix, which returns a list of feature subsets that have high correlation with each other. It calculates the mean of the columns in each subset and stores the result in the first column of the subset.It drops the other columns in each subset from the DataFrame data. Finally, it returns the DataFrame with reduced features.

Lastly, `DataProcess.linearfeatures()` iterates through all features and checks the correlation coefficient with every other feature. If the absolute difference in the correlation between two features is less than 0.11 and their correlation coefficient is

greater than or equal to 0.95, it groups them together as linear features. Once all linear features have been identified, the function combines intersecting groups into a single set to prevent overlap between features. Finally, the function returns a list of linear feature sets, where each set represents a group of features that are highly cross-correlated with each other.

To be more specific, `DataProcess.crosscor()` calculates the pairwise correlation between each pair of features, and selects the pairs (a,b) that have a correlation of >0.95 and a similar correlation profile when with other features. If corr(a,c) and corr(b,c) have similar values for all other features c, then we can combine them. An example of two such features would be 'currentPrice' and 'targetHighPrice' as they almost effectively move in sync on a daily basis.

Predictions

We then call `Predict()`, which performs the bulk of the rule induction. The function selects the outputs to predict (these are either provided, or the sample array `ycol` is used). The function then calls the `DataProcess.crosscorr()`, and calculates the sorted and unsorted cross-correlations using the Spearman method. The `DataProcess.pickfeatures()` function is called next, which selects the top 12 features based on their correlation values and a threshold of 0.5. These features are stored in the features variable.

Next, the `data` dataframe is trimmed by the tickers sector and market cap, so that stocks only with the same market capitalisation and sector are selected. This is to significantly speed up the KNN calculations. The assumption is that the best KNNs are likely to have a similar market capitalisation and sector, which is not unreasonable.

In the k-nearest neighbour step, the function uses three different distance metrics (Euclidean, Minkowski, and cosine) to find the k-nearest neighbours of the input ticker. These neighbours are then combined in Union. After this, we calculate the discretised data using `DataProcess.discretisation()`, which involves turning the data from continuous to discrete. This is necessary so that trivial rulesets, like those that rely on one continuous variable, are avoided. The cleaned dataset is checked to ensure that there are no NaN values, and the discrete dataset is prepared for rule induction.

The `RuleInduction.main()` module is called to induce rules from the discretised dataset. This function returns the minimal covering set, and if a minimal covering set

cannot be found or the rule induction is not possible due to sparseness of data from YahooQuery, an appropriate message is printed.

Finally, the efficiency of the induced rules is calculated by subtracting the number of rules in the minimal covering set from the total number of attributes in the discretized dataset and dividing by the total number of attributes. The efficiency ranges from 0 to 1, where 1 indicates that the induced rules cover all instances and do so using a minimal number of attributes. The output of the function is the minimal covering set, the induced rules, the efficiency, and a flag indicating if the function was successful or not. The flag is set to True if the rule induction is not possible due to sparseness of data from YahooQuery.

Final Generation

Lastly, the final calculations are done using the specified functions above. We first prune the features using `PruneFeatures()`, which works as described above. Once this is done, we set a timeout of 5 minutes, and attempt to general a working ruleset repeatedly using `Predict()`. Specifically, we alter the number of k-nearest neighbours and number of discrete sections for discretisation on each run, to ensure each run is unique. If we manage to create a working ruleset using `Predict()`, our program outputs the ruleset.

How does Rule Induction Work?

Given a sufficiently large dataset to work from, we aim to infer associative rules which can uniquely and accurately describe the behaviour of datapoints locally. To do this, we implemented the LEM1 Algorithm, a component of the data mining algorithm LERS (Learning from Examples using Rough Sets)[1].

First, we define the elementary set function given a set of attributes and items(cases/datapoints). An elementary set *B* is a unique collection of sets, where *x, y* belong in the same set if and only if they have equivalent values across all the input attributes. As an example, consider the following table below:

[1] For more information on this, see the book excerpt covering Rule Induction by J.W. Grzymala-Busse below - https://people.eecs.ku.edu/~jerzygb/Rule-Induction-new.pdf. This excerpt defines LEM1 formally, explaining the mathematical basis for the algorithm.

Case	Headache	Temperature	Age
1	yes	high	Child
2	no	very_high	Teenager
3	yes	normal	Adult
4	yes	normal	Adult
5	no	very_high	Teenager

The attributes here are {Headache, Temperature, Age}. The elementary sets for {Headache, Temperature} would be {{1}, {2,5}, {3,4}}. As observed, the equivalent inputs are gathered together into unique sets.

Next, we define `flatten()`, which recursively flattens a list, and `consistent(B,D)`, which checks if an elementary set `B` is consistent with an elementary set `D`. Specifically, the set `B` being consistent with `D` means that every set in B is a subset of at least one set in `D`. For example, consider `B` ={{1}, {2}, {3,4}, {5}}. This is consistent with D= {{1}, {2,5}, {3,4}}, and we can write $B \leq D$. On the other hand, if `B` = {{1}, {2}, {3,5}, {4}}, then `B` is not consistent with `D` because {3,5} is not a subset of {3,4} or {2,5} in `D`.

Next, we define `smallestcover()`, which finds the smallest set of features that can uniquely describe the covering set. In the table above, the column 'Age' can uniquely predict the values of {Headache, Temperature}, but it is unlikely that a covering would be this trivial in larger sets of data. It should also be noted that the data fed into this function is discretised, so that continuous data which is unique for each stock (such as, say, the stock price) does not result in a trivial covering.

Next, we define `inferrule()`, which takes in data, a subcover (as defined by `smallestcover()`), and a list of decisions). First, it creates a collection of sets `B` that agree on all features in the subcover. Then, for each set in `B`, it iteratively drops features that do not contribute to the set being a pure set (i.e., all samples in the set have the same decision output). It creates a rule for the pure set by using the remaining features as conditions and the decision outputs as the decision. The rules for all pure sets are stored in a dictionary `RuleSet` and returned.

Lastly, we define `main()`, which is the function used in `main.Predict()`. First, it creates a collection of elementary sets `B` and `D` for the attributes and decision outputs, respectively. Then, it checks if the dataset is consistent using the `consistent()` function. If the dataset is consistent, it finds the smallest covering set of features for the decision outputs using the `smallestcover()` function. If the function successfully finds a minimal covering set and it is smaller than the original set of attributes, it calls the `inferrule()` function to generate a set of rules from the data. Otherwise, it returns `False` for both the minimal covering and the rules.

K Nearest Neighbours Calculations

The code also includes KNN.py, which is an implementation of the k nearest Neighbours algorithm `kNearest()` outputs a dictionary where the keys are the indices of the rows in the dataframe, and the values are a list of tuples representing the k nearest neighbours of that row, along with their distances. This function is dependent on two other functions - `cdist()`, which computes the pairwise distances between all rows in a dataframe, and `distance()`, which takes two vectors and computes the distance between them based on the specified metric (e.g. euclidean).

Example Output

Using the following code input for parameters:

```
ticker = "IBM"
output = ['absVaR', 'sharpe']
```

The output is:

```
Success: Local-Rule estimation of stock IBM
Nearest Neighbours are:  ['VRNT', 'DELL', 'DUK', 'ADP', 'UNP',
'GILD', 'ACN', 'UPS', 'ADI', 'MDT', 'WM', 'MO', 'CSCO', 'AMGN',
'MMM', 'PM', 'SO', 'SWKS', 'APD', 'SC>
A locally minimal covering set of features with efficiency
0.7333333333333334  is:  ['bookValue', 'forwardEps',
'marketCap', 'fiftyTwoWeekHigh']
A local covering set of Rules is:
(('marketCap', 2), ('fiftyTwoWeekHigh', 2)) ----> [('absVaR',
1), ('sharpe', 4)]
(('forwardEps', 4), ('marketCap', 3), ('fiftyTwoWeekHigh', 2))
----> [('absVaR', 2), ('sharpe', 6)]
(('forwardEps', 4), ('marketCap', 4), ('fiftyTwoWeekHigh', 4))
----> [('absVaR', 0), ('sharpe', 4)]
(('forwardEps', 5), ('marketCap', 5), ('fiftyTwoWeekHigh', 6))
----> [('absVaR', 0), ('sharpe', 6)]
(('bookValue', 3), ('forwardEps', 6)) ----> [('absVaR', 0),
('sharpe', 5)]
(('forwardEps', 4), ('marketCap', 5), ('fiftyTwoWeekHigh', 3))
----> [('absVaR', 1), ('sharpe', 4)]
('fiftyTwoWeekHigh', 7) ----> [('absVaR', 0), ('sharpe', 6)]
(('forwardEps', 6), ('fiftyTwoWeekHigh', 5)) ----> [('absVaR',
0), ('sharpe', 5)]
(('marketCap', 5), ('fiftyTwoWeekHigh', 5)) ----> [('absVaR',
1), ('sharpe', 6)]
(('bookValue', 5), ('fiftyTwoWeekHigh', 3)) ----> [('absVaR',
0), ('sharpe', 4)]
(('forwardEps', 4), ('marketCap', 4), ('fiftyTwoWeekHigh', 5))
----> [('absVaR', 0), ('sharpe', 8)]
(('marketCap', 5), ('fiftyTwoWeekHigh', 2)) ----> [('absVaR',
0), ('sharpe', 4)]
('marketCap', 8) ----> [('absVaR', 0), ('sharpe', 5)]
('forwardEps', 8) ----> [('absVaR', 0), ('sharpe', 5)]
```

```
(('forwardEps', 5), ('marketCap', 4), ('fiftyTwoWeekHigh', 4))
----> [('absVaR', 0), ('sharpe', 3)]
('bookValue', 1) ----> [('absVaR', 0), ('sharpe', 3)]
(('forwardEps', 3), ('marketCap', 5)) ----> [('absVaR', 0),
('sharpe', 5)]
(('marketCap', 3), ('fiftyTwoWeekHigh', 4)) ----> [('absVaR',
2), ('sharpe', 5)]
('fiftyTwoWeekHigh', 8) ----> [('absVaR', 0), ('sharpe', 6)]
(('marketCap', 4), ('fiftyTwoWeekHigh', 3)) ----> [('absVaR',
1), ('sharpe', 5)]
(('marketCap', 7), ('fiftyTwoWeekHigh', 3)) ----> [('absVaR',
0), ('sharpe', 7)]
(('forwardEps', 3), ('marketCap', 3), ('fiftyTwoWeekHigh', 2))
----> [('absVaR', 1), ('sharpe', 5)]
('bookValue', 0) ----> [('absVaR', 1), ('sharpe', 6)]
(('marketCap', 4), ('fiftyTwoWeekHigh', 6)) ----> [('absVaR',
0), ('sharpe', 6)]
(('forwardEps', 3), ('marketCap', 3), ('fiftyTwoWeekHigh', 3))
----> [('absVaR', 2), ('sharpe', 4)]
(('forwardEps', 3), ('marketCap', 2), ('fiftyTwoWeekHigh', 3))
----> [('absVaR', 2), ('sharpe', 5)]
(('forwardEps', 4), ('marketCap', 3), ('fiftyTwoWeekHigh', 3))
----> [('absVaR', 1), ('sharpe', 4)]
(('forwardEps', 5), ('fiftyTwoWeekHigh', 3)) ----> [('absVaR',
0), ('sharpe', 4)]
(('forwardEps', 4), ('fiftyTwoWeekHigh', 6)) ----> [('absVaR',
0), ('sharpe', 5)]
(('marketCap', 6), ('fiftyTwoWeekHigh', 4)) ----> [('absVaR',
0), ('sharpe', 4)]
(('forwardEps', 4), ('marketCap', 6), ('fiftyTwoWeekHigh', 5))
----> [('absVaR', 1), ('sharpe', 6)]
(('forwardEps', 5), ('marketCap', 6), ('fiftyTwoWeekHigh', 6))
----> [('absVaR', 0), ('sharpe', 5)]
(('forwardEps', 5), ('marketCap', 4), ('fiftyTwoWeekHigh', 5))
----> [('absVaR', 1), ('sharpe', 5)]
(('forwardEps', 4), ('marketCap', 2)) ----> [('absVaR', 3),
('sharpe', 5)]
Parameters used are: 8 intervals for data discretisation and
K=[35, 30, 5]
```