Financial Time Series Forecasting via Graph Neural Networks

Candidate Number: 1013530

Word Count: 7,499 (TeXcount)

University of Oxford

A thesis submitted for the degree of

Mathematics and Statistics Part C

Trinity 2020

Abstract

In the current stock price forecasting literature there exists a large body of work devoted to multivariate techniques which leverage the dependencies between individual time series to produce a prediction. Despite the strengths of these methods, they fail to take into account the network architecture displayed by the total market. We introduce a novel multivariate forecast strategy called TemPr, that uses graph neural networks (GNNs) to uncover low-dimensional representations of the stock market over time, and tracks changes in this temporal structure. Additionally, unlike existing methods, TemPr incorporates volume and beta time series for each stock in its forecasting pipeline. We also present the univariate sibling of this method, UniTempr, which harnesses the power of GNNs to compare sub-intervals of a time series in order to produce a forecast. The power in taking the multivariate network approach is demonstrated in our results where we back-test these techniques from 2007-2019, comparing their Sharpe ratios and P&L scores. TemPr achieves a Sharpe ratio above 2.5, improving on the performance of LSTM networks, and significantly outperforms UniTempr, further supporting our hypothesis that it is essential to consider the temporal state of the entire market when producing forecasts.

Keywords: multivariate time series, graph neural networks, stock market, correlation networks

CONTENTS

ACRONYMS

ARIMA Auto-Regressive Integrated Moving Average. 1, 2, 29

BFS Breadth First Sampling. 14

DEIC The Dutch East India Co.. 1, 2

DFS Depth First Sampling. 14

DTW Dynamic Time Warping. 7

GCN Graph Convolutional Network. 11

GNN Graph Neural Network. 2, 8, 9, 45

KNN K-Nearest Neighbours. 2, 8, 21

LSTM Long Short-Term Memory. 1, 2, 11, 27, 32, 42

NLP Natural Language Processing. 42

PA Procrustes Analysis. 16

PnL Profit and Loss. 2, 24, 39

PPT PnL Per Trade. 2, 25, 39, 43

RNN Recurrent Neural Network. 42

S&P Standard and Poor's. 3

SR Sharpe Ratio. 23, 24, 36, 37

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Stock markets have long been the financial backbone of the world's economies, dating back to the late 1500s. In 1602, The Dutch East India Co. (DEIC) became the first publicly traded company by issuing shares to the general public [21]. As in modern stock markets today, risk was the driving factor at the heart of this decision. When the East Indies were first colonised by the Dutch and found to be a haven of natural resources and trade, explorers from across the world sailed in hordes to exploit its many riches. It was an extremely dangerous sailing route with pirates running rife in the Indian Ocean, so few of these voyages ever made it back. Investors who were funding these trips realised investing in a single voyage, and therefore 'placing all their eggs in one basket', was not financially viable and needed a strategy to mitigate this risk. Motivated by this, DEIC released shares of the company on the Amsterdam Stock Exchange with investors entitled to a fixed percentage of their profits. It proved to be an extremely lucrative strategy. Instead of investing in one voyage, investors could now purchase shares from a portfolio of companies making the trip, meaning even if one ship did not make it back they would still make a profit.

Stock markets have greatly evolved since then, yet risk is still the common fundamental theme, now deriving from whether your investment portfolio behaves as your models predicted, rather than your ship sinking. With the advent of digital trading and modern computing power, the task of mitigating this risk has turned to the technological realm. It is now common practice for investment banks and hedge funds to have strategies computed by complex algorithms harnessing huge quantities of data from the market. The field of stock-price time series forecasting has seen particular growth over the last few decades with the development of models such as Auto-Regressive Integrated Moving Average (ARIMA) [3], and in more recent years Long

Short-Term Memory (LSTM) [6] neural networks. These strategies predict whether each stock-price will rise or fall over the coming days, and hence whether to hold a long or short position respectively.

These methods can be implemented in a multivariate context, as demonstrated with LSTM by Torres et al. [9], but fail to show significant improvements on traditional forecasting models. Whilst they harness information from the dependencies between assets, they seldom consider the network structure of connections formed between each stock's time series and the temporal state of the total market. It makes sense to consider these connections even in the context of the DEIC, where investors would be unlikely to buy a share in a historically successful ship if many of the ships that sail the same route had recently been sunk by pirates. Furthermore, modern forecasting models rarely take features other than stock-returns as inputs. Our techniques will overcome these two key issues in market forecasting literature by considering multivariate time series as a graph-structured data-set with node level attributes of volume and beta.

1.2 Contribution

Within Chapter 3, we present a stock-return forecasting technique called TempPr. TemPr uses Graph Neural Networks (GNN) to generate a low dimensional embedding of the market for every past trading day and tracks the temporal changes in these daily embeddings over time. It outputs a forecast for each stock by comparing the market's current temporal state to historically similar days using a K-Nearest Neighbours (KNN) algorithm. We discuss a variety of methods that generate such embeddings including DeepWalk [18], node2vec [11] and the GNN architectures of GraphSAGE [12]. We compare using each embedding method in the proposed TempPr pipeline and select the most successful, by assessing their forecast performance using PnL, PPT and Sharpe Ratio metrics. Then, after further parameter tuning of the final TemPr method, we see that this novel temporal approach to forecasting outperforms state-of-the-art models such as ARIMA [3], LSTM [6] and Facebook's Prophet [23].

In Chapter 4 we introduce a univariate form of TemPr that we call UniTemPr, which uses a GNN-generated embedding of a stock's history to produce forecasts. We draw a parallel between our GNN-based methods and LSTM networks, which both use long-term dependencies within stocks to generate forecasts. We assess the results and see that UniTemPr's performance is on par with state-of-the-art univariate methods. However, it is still outperformed by TemPr and other multivariate methods highlighting the strength in considering the total network structure of the market.

Within all chapters, we utilise data consisting of S&P1500 constituents spanning from 28/01/2000 until 19/07/2019 to distinguish the performance of our techniques, specifically using the log-adjusted close price, volume, and beta to the market ETF 'SPY'.

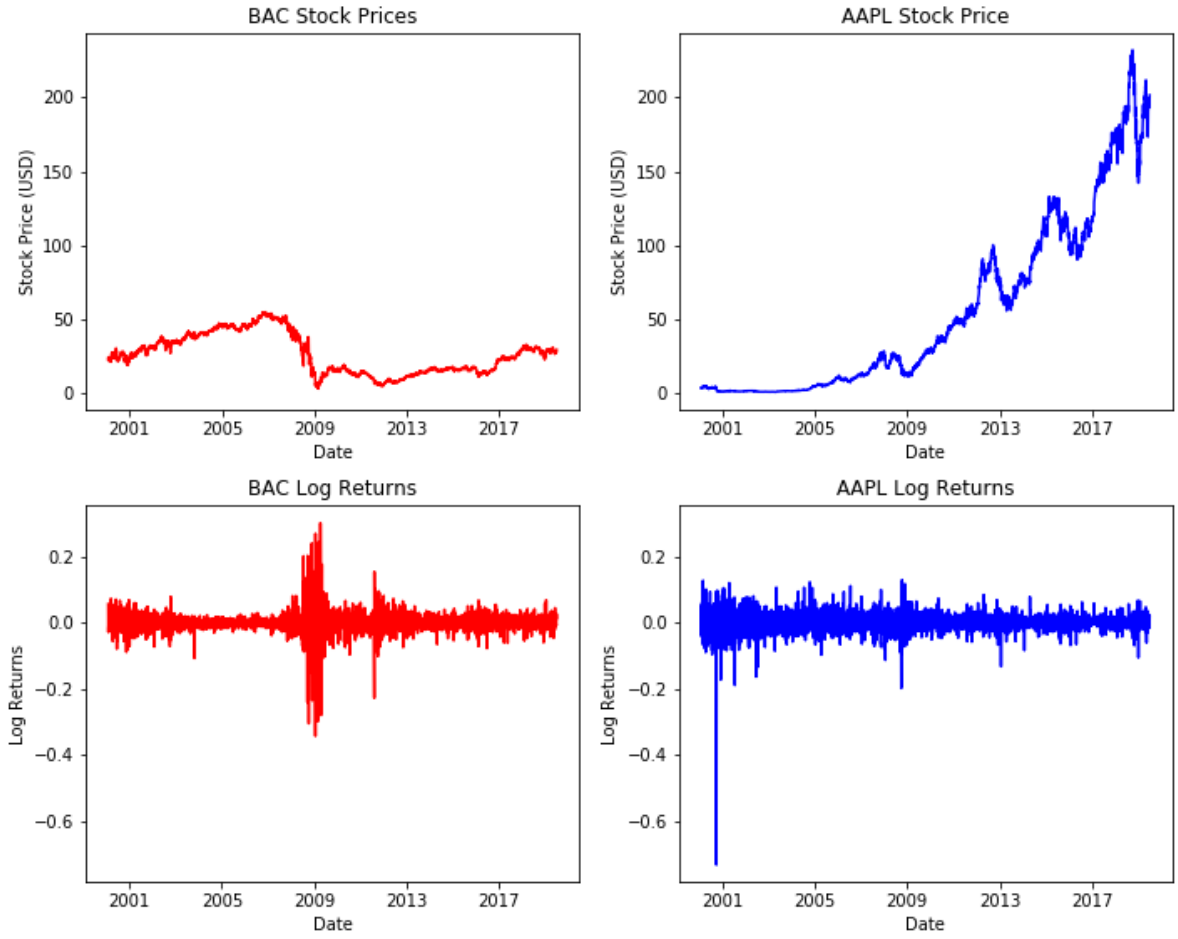


Figure 1.1: Comparison of raw stock prices and log-adjusted return ratios for Bank of America (BAC) and Apple (AAPL) instruments.

CHAPTER 2

THEORETICAL FRAMEWORK

There is a library of mathematical concepts deployed throughout this project which we address in this chapter, chronologically to how they are employed in the TemPr pipeline.

2.1 Time Series Correlation Measures

The raw data we collect from the stock market is in the form of N stock price time series over the last T days. We denote time series of length $m \leq T$ for the returns of stocks X and Y by:

$$X = (x_1, x_2, ..., x_m) \text{ and } Y = (y_1, y_2, ..., y_m). \tag{2.1}$$

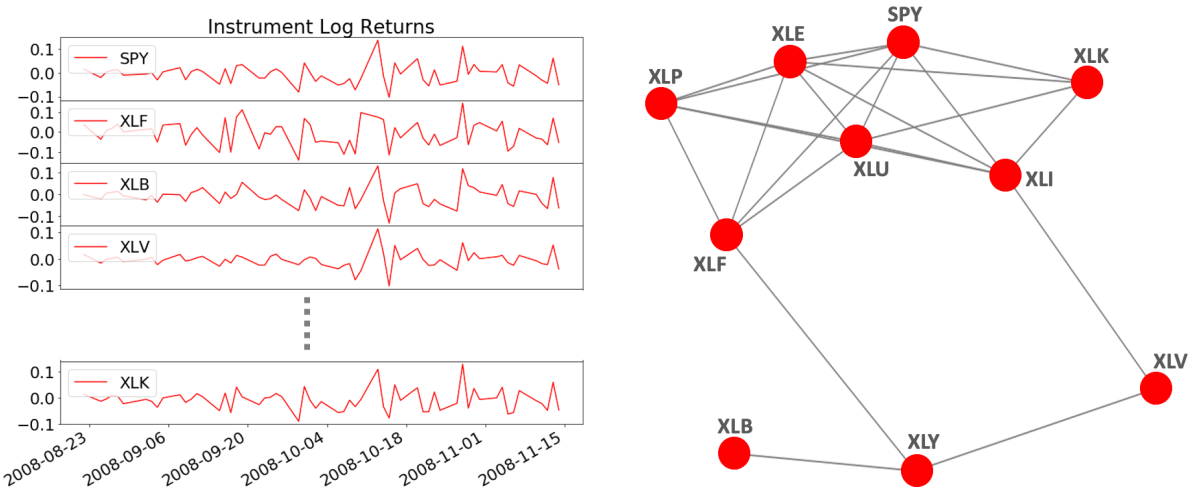
In order to create a network structure between the stocks, we must generate an adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, where each element $\mathbf{A}_{XY}$ corresponds to the correlation between a pair (X, Y) of time series. These entries are thresholded by some $c \geq 0$ to guarantee non-negativity and to adjust the sparsity of the resulting network. We treat the correlations as edge-weights of a connected graph, with vertices corresponding to stocks. We present three measures that calculate this pairwise similarity $\mathbf{A}_{XY}$.

2.1.1 Pearson Correlation Coefficient

The Pearson correlation coefficient, ρ, is a measure of the linear dependence between two variables. For time series X and Y, their *Pearson correlation* is defined as:

$$\rho_{XY} \coloneqq \frac{\mathrm{Cov}(X, Y)}{\sqrt{\mathrm{Var}(X)}\sqrt{\mathrm{Var}(Y)}}. \tag{2.2}$$

Thus, we take $\mathbf{A}_{XY} = \rho_{XY}$ as entries in adjacency matrix $\mathbf{A}$.



(a) 60-day log returns time series for 10 instruments **(b)** Example network of same ETFs

Figure 2.1: Example of converting multivariate time series to a network on 10 instruments from S&P1500 with an edge-weight threshold of 0.4

2.1.2 Distance Correlation

Distance Correlation [22] is a measure of the statistical dependence between two random variables and can detect both linear and non-linear association. This can make it more adept for quantifying relationships between stocks than Pearson Correlation.

Define distance matrices $\mathcal{A}, \mathcal{B} \in \mathbb{R}^{m \times m}$ where $\mathcal{A}_{ij} = ||x_i - x_j||_1$ and $\mathcal{B}_{ij} = ||y_i - y_j||_1$, for $i, j = 1, .., m$. Then double centre the matrices $\mathcal{A}, \mathcal{B}$ to get $\widetilde{\mathcal{A}}, \widetilde{\mathcal{B}}$ where:

$$\widetilde{\mathcal{A}}_{ij} = \mathcal{A}_{ij} - \bar{\mathcal{A}}_{i\cdot} - \bar{\mathcal{A}}_{\cdot j} + \bar{\mathcal{A}}_{\cdot\cdot}$$
$$\widetilde{\mathcal{B}}_{ij} = \mathcal{B}_{ij} - \bar{\mathcal{B}}_{i\cdot} - \bar{\mathcal{B}}_{\cdot j} + \bar{\mathcal{B}}_{\cdot\cdot} \, .$$

Here we denote $\bar{\mathcal{A}}_{i\cdot}$ as the mean of row i, $\bar{\mathcal{A}}_{\cdot j}$ the mean of column j, and $\bar{\mathcal{A}}_{\cdot\cdot}$ the mean of matrix $\mathcal{A}$. The squared *distance covariance* of X, Y is given by:

$$dCov_n^2(X, Y) := \frac{1}{m^2} \sum_{i,j=1}^{m} \widetilde{\mathcal{A}}_{ij} \widetilde{\mathcal{B}}_{ij} \, . \tag{2.3}$$

Then, the *distance correlation* of X, Y is defined as:

$$dCor_n^2(X, Y) := \frac{dCov_n^2(X, Y)}{\sqrt{dCov_n^2(X, X)} \sqrt{dCov_n^2(Y, Y)}}. \tag{2.4}$$

This takes values in the range $[0, 1]$, with $dCor_n^2(X, Y) = 0$ implying independence between X and Y. We then take entries of adjacency as $\mathbf{A}_{XY} = dCor_n^2(X, Y)$.

2.1.3 Dynamic Time Warping

Dynamic Time Warping (DTW) [17] is a metric that computes the dissimilarity between sequences moving at different speeds. Let $\mathbf{M} \in \mathbb{R}^{m \times m}$ be the matrix of point-to-point L^1-distances between time series X and Y, where element $\mathbf{M}_{ij} = ||x_i - y_j||_1$. Define a path that traverses the matrix from entry $\mathbf{M}_{11}$ to $\mathbf{M}_{mm}$ as $P = (p_1, p_2, ...p_K)$, where each element $p_k = (i, j)$ indicates the two points the path connects at step k, and $m < K < 2m - 1$. Then, the *DTW distance* between the two time series is the weight of the lowest cost path:

$$DTW(X, Y) := \min_P \left\{ \sum_{k=1}^{K} d_k : P = (p_1, p_2, ...p_K) \right\}, \tag{2.5}$$

where $d_k = ||x_i - y_j||_1 = \mathbf{M}_{ij}$ is the distance between the points that p_k connects. This measure takes values in $\mathbb{R}^{\geqslant 0}$. In our adjacency matrix $\mathbf{A}$, we take the reciprocal as entries $\mathbf{A}_{XY} = [DTW(X, Y)]^{-1}$ in order to give higher weight to similar time series.

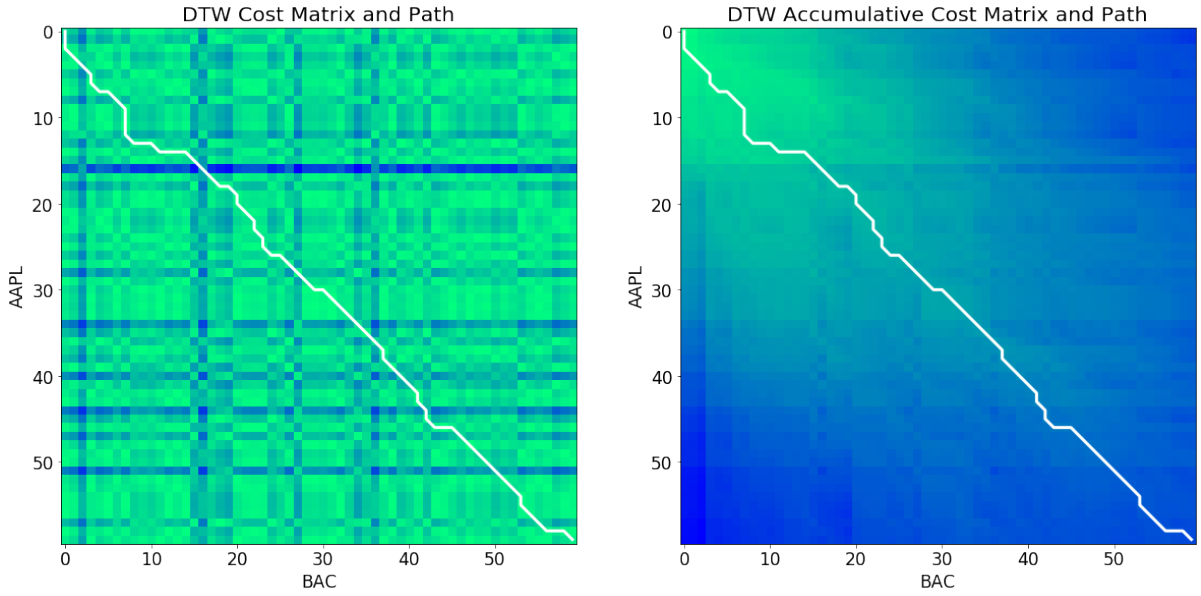


Figure 2.2: DTW path P between AAPL and BAC instruments over the 60 days from 13/11/2000 to 08/02/2001, with lighter colours implying lower weight. The left figure represents the cost of each entry in the weight matrix $\mathbf{M}$, and the right is the accumulative cost of the path

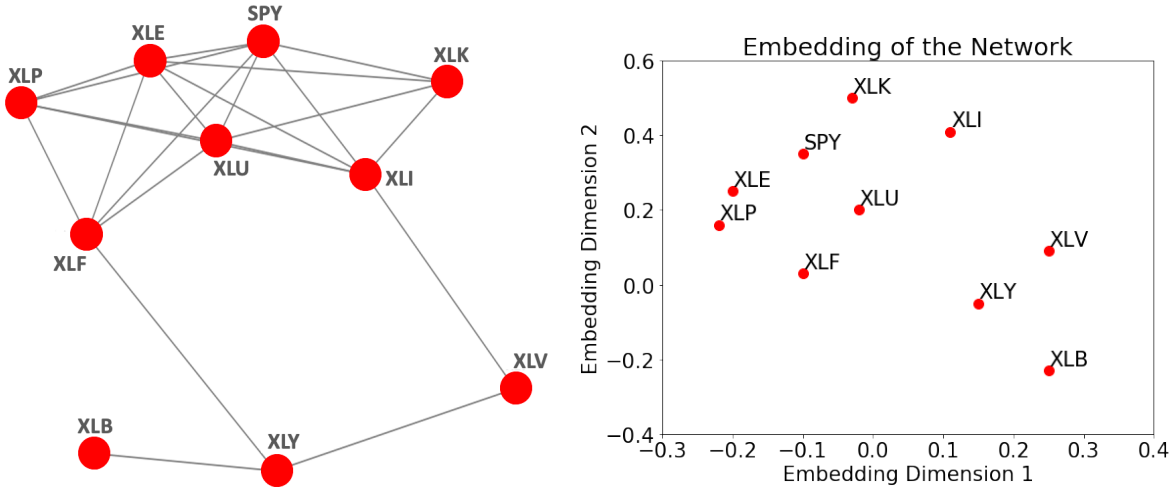
2.2 Network Embedding Techniques

A crucial step in the TemPr pipeline uses GNNs to uncover a low-dimensional embedding of the graph structured stock-market, as we can then leverage these embeddings for temporal tracking and downstream forecasting techniques, such as KNN. These embedding methods aim to compute a representation of the similarity between nodes as a distance in a vector space.

Hence, for a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the goal for these models is to learn a function of signals and features which takes as inputs the following:

- Representative description of the graph structure from adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, as outlined in 2.1.
- Feature description for every node collected in matrix $\mathbf{F} \in \mathbb{R}^{N \times d}$, where d is the number of input features.

These models produce a node-level output matrix $\mathbf{Z} \in \mathbb{R}^{N \times D}$ where D is the embedding dimension. We now discuss two frameworks to generate such embeddings.



(a) Input Network of Market Instruments **(b)** Output Embedding of Sample Network

Figure 2.3: Example of 2D network embedding on 10 Instruments from S&P1500

2.2.1 Graph Neural Networks

The following GNNs are based on neighbourhood aggregation schemes, where the low-dimensional representation vector of a node is computed by recursively aggregating and transforming representation vectors of neighbouring nodes. Kipf & Welling [15] proposed using a multi-layer neural network to recursively aggregate the features from each node's neighbourhood.

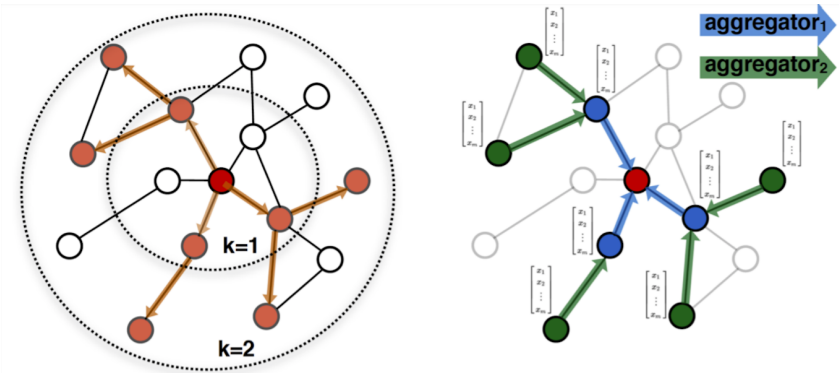


Figure 2.4: GraphSAGE - Neighbourhood exploration and feature aggregation [12].

2.2.1.1 Algorithmic Structure

GNNs all follow the same neural network structure, given by Algorithm 1 [12], and differ only in their aggregation techniques which we discuss in Section 2.2.1.2. We say that this method is an inductive learning algorithm as it learns embedding functions that can be applied to unseen nodes rather than having to re-train the whole algorithm. The algorithm can be addressed in two steps:

Step 1: Lines 1-7 sample the nodes needed for aggregating the neighbourhoods of minibatch $\mathcal{B}$ to depth L, on graph $\mathcal{G}$. At each step l, the process generates a neighbourhood set $\mathcal{N}(u) \subseteq \mathcal{V}, \forall u \in \mathcal{B}^l$ and is repeated until we have taken L steps from the set $\mathcal{B}$. This uses uniform neighbourhood function $\mathcal{N} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$ where $\mathcal{N}(u)$ is defined as a fixed-size uniform draw from the set $\{v \in \mathcal{V} : (u, v) \in \mathcal{E}\}$ [12]. This forward propagation process enables us to employ stochastic gradient descent in training to minimise our loss function, as described in Section 2.2.1.3.

Step 2: Lines 9-16 describe the aggregation process. For node u, the embedding vector $\mathbf{z}_u$ is generated by inputting feature vector $\mathbf{F}_u = \mathbf{h}_u^0$ into the first layer and then recursively forward passing it through each of the L layers of the network to eventually give $\mathbf{z}_u = \mathbf{h}_u^L$.

At step l for node $u \in \mathcal{V}$, line 11 collects the current representations of the nodes from u's immediate neighbourhood $\mathcal{N}_l(u)$ and aggregates into the vector $\mathbf{h}_{\mathcal{N}(u)}^l$. This is concatenated with u's current representation $\mathbf{h}_u^{l-1}$ and then fed through a dense layer with activation σ_R. After normalising, we arrive at the updated representation $\mathbf{h}_u^l$. The intuition is that at each step l in the outer loop, nodes aggregate more information from deeper depths of their neighbourhoods.

Algorithm 1: Graph Neural Network Minibatch Embedding Algorithm

Inputs: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$; minibatch $\mathcal{B} \subseteq \mathcal{V}$; input features $\{\mathbf{F}_u, \forall u \in \mathcal{V}\}$; depth L; weight matrices $\mathbf{W}^{(l)}, \forall l \in \{1, ..., L\}$; differentiable aggregator functions $AGGREGATE_l, \forall l \in \{1, .., L\}$; uniform neighbourhood function $\mathcal{N} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$; ReLU activation function $\sigma_R(\cdot) = max\{0, \cdot\}$

Outputs: Vector representations $\mathbf{z}_u \in \mathbb{R}^D, \forall u \in \mathcal{B}$

1 Set $\mathcal{B}^L \leftarrow \mathcal{B}$;
2 **for** $l = L, .., 1$ **do**
3 $\mathcal{B}^{l-1} \leftarrow \mathcal{B}^l$;
4 **for** $u \in \mathcal{B}^l$ **do**
5 $\mathcal{B}^{l-1} \leftarrow \mathcal{B}^{l-1} \cup \mathcal{N}_l(u)$;
6 **end**
7 **end**
8 $\mathbf{h}_u^0 \leftarrow \mathbf{F}_u, \forall u \in \mathcal{B}^0$;
9 **for** $l = 1, ..., L$ **do**
10 **for** $u \in \mathcal{B}^l$ **do**
11 $\mathbf{h}_{\mathcal{N}(u)}^l \leftarrow AGGREGATE_l\big(\{\mathbf{h}_v^{l-1}, \forall v \in \mathcal{N}_l(u)\}\big)$;
12 $\mathbf{h}_u^l \leftarrow \sigma_R\Big(\mathbf{W}^{(l)} \cdot CONCAT\big(\mathbf{h}_u^{l-1}, \mathbf{h}_{\mathcal{N}(u)}^l\big)\Big)$;
13 $\mathbf{h}_u^l \leftarrow \mathbf{h}_u^l / ||\mathbf{h}_u^l||_2$;
14 **end**
15 **end**
16 **Return:** $\mathbf{z}_u \leftarrow \mathbf{h}_u^L \in \mathbb{R}^D, \forall u \in \mathcal{B}$

2.2.1.2 Aggregator Architectures

The aggregation of neighbour representations can be implemented with the following range of architectures.

Graph Convolutional Network Kipf & Welling [15] first proposed using the multi-layered neural-network approach to learning graph representations, where they aggregate neighbouring nodes' information by taking the weighted mean of their feature representations. The weighting is proportional to the edge-weight between the node and its neighbour given by adjacency matrix $\mathbf{A}$. Diagonal entries of $\mathbf{A}$ are set equal to 1 so that the node under question is included in the mean calculation itself. We arrive at the algorithm they derived by replacing lines 11 & 12 in Algorithm 1 with:

$$\mathbf{h}_u^l \leftarrow \sigma_R\left(\mathbf{W}^{(l)} \cdot \frac{1}{\hat{\mathbf{A}}_{u\cdot}} \sum_{v \in \mathcal{N}_l(u) \cup \{u\}} \mathbf{A}_{uv} \cdot \mathbf{h}_v^{l-1}\right), \tag{2.6}$$

where $\hat{\mathbf{A}}_{u\cdot} = \sum_{v \in \mathcal{N}_l(u) \cup \{u\}} \mathbf{A}_{uv}$ denotes the sum of the edge-weights between u and its neighbours $v \in \mathcal{N}_l(u) \cup \{u\}$.

GraphSAGE Mean The GraphSAGE mean aggregator is very similar to the GCN variant. The main difference is that GraphSAGE excludes the node under question from the mean calculation and instead concatenates its representation with the mean neighbourhood representation. This concatenation step can be thought of as a "skip connection" [13] between different layers of the neural network structure and leads to great embedding performance improvement [12]. Thus, for GraphSAGE Mean, the aggregator function in line 11 is defined by:

$$AGGREGATE_l^{MEAN} := \frac{1}{\tilde{\mathbf{A}}_{u\cdot}} \sum_{v \in \mathcal{N}_l(u)} \mathbf{A}_{uv} \cdot \mathbf{h}_v^{l-1}, \tag{2.7}$$

where $\mathbf{A}$ is the adjacency matrix of graph $\mathcal{G}$, and $\tilde{\mathbf{A}}_{u\cdot} = \sum_{v \in \mathcal{N}_l(u)} \mathbf{A}_{u,v}$ denotes the sum of the edge weights between u and its neighbours $v \in \mathcal{N}_l(u)$.

GraphSAGE Pooling In the GraphSAGE pooling aggregators, neighbour vector representations are each passed through a fully connected single-layered neural network. Then, the element-wise maximum or mean is calculated from the outputs of this network. Thus, the GraphSAGE Maxpool and Meanpool aggregator functions are given by:

$$AGGREGATE_l^{MAXPOOL} := \max \Big\{ \sigma\big(\mathbf{W}_{pool} \cdot \mathbf{h}_v^l + \mathbf{b}\big), \quad \forall v \in \mathcal{N}_l(u) \Big\}, \tag{2.8}$$

$$AGGREGATE_l^{MEANPOOL} := \frac{1}{|\mathcal{N}_l(u)|} \sum_{\forall v \in \mathcal{N}_l(u)} \sigma\big(\mathbf{W}_{pool} \cdot \mathbf{h}_v^l + \mathbf{b}\big), \tag{2.9}$$

where $\mathbf{W}_{pool}$ and $\mathbf{b}$ are extra parameters that are learnt when minimising the loss function, and σ is a nonlinear activation function.

GraphSAGE LSTM GraphSAGE LSTM employs a LSTM cell as the aggregator function [12] which we discuss in depth in Section 3.1.6. Aggregators are required to be symmetric so that they are invariant under permutation of neighbouring nodes, otherwise the order that the neighbours are fed into the LSTM would impact the embedding. LSTMs are not naturally symmetric as they are designed for sequential operation and have artificial memory. To overcome this, Hamilton et al. [12] suggested feeding the neighbouring nodes into the LSTM in a random permutation. As with the pooling aggregators, the parameters are learnt during loss minimisation.

2.2.1.3 Parameter Tuning

We employ a semi-supervised graph-based loss function to learn the weight matrices $\mathbf{W}^l, \forall l \in \{1, ..., L\}$ used to generate the node embeddings. With this loss function, each embedding $\mathbf{z}_u \in \mathbb{R}^D, \forall u \in \mathcal{V}$, has the associated loss [12] :

$$L(\mathbf{z}_u) = -\log\Big(\sigma_S(\mathbf{z}_u^\intercal \mathbf{z}_v)\Big) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\big(\sigma_S(\mathbf{z}_u^\intercal \mathbf{z}_{v_n})\big), \tag{2.10}$$

where $v \in \mathcal{N}(u)$, $\sigma_S(x) = (1 + e^{-x})^{-1}$ is the logistic sigmoid function, P_n is a negative sampling distribution, v_n is a negative sample drawn from P_n, and Q is the number of such negative samples. In this context, a negative sample represents nodes that are not neighbours of u. Intuitively, the first term promotes maximising the similarity of the embeddings of u and neighbour v, whilst the second term aims to minimise the similarity of u and the negative sample [25]. We use stochastic gradient descent to search for the parameters that represent the minimum loss.

2.2.2 DeepWalk and node2vec

In contrast to GNNs, DeepWalk and node2vec follow an embedding framework that aggregates information from the local structure of the graph by taking random walks from each node. It is based on the assumption that adjacent nodes in these walks should have similar embeddings.

2.2.2.1 Algorithmic Structure

Both of these methods employ the same algorithmic structure which we present in Algorithm 2 [11][18] and only differ in the way that they generate random walks. Algorithm 2 can be broken down into two steps:

Step 1: In lines 1-12, the algorithm traverses the graph with random walks to infer the local structure for each node. The two inner loops generate a random walk of length L, starting from each node u. The "$SAMPLE$" function picks the next node to visit in the walks and we discuss how this differs between DeepWalk and node2vec in Section 2.2.2.2. The outer loop then repeats this process until we have r such random walks for each node.

Step 2: In lines 13-21, the algorithm use the SkipGram model [18] to optimise embeddings using the graph context provided by the walks. SkipGram starts by initialising the embeddings to random vectors in $\mathbb{R}^D$. It then iterates over every node in each walk, optimising at each step separately using gradient descent. This updates the embeddings in order to maximise the probability of the neighbours of a node, given the node itself. This method is called transductive as it requires the entire graph to train the embeddings.

Algorithm 2: node2vec Embedding Algorithm

Inputs: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$; embedding dimension D; walks per node r; walk length L; context size k; neighbourhood generating function $\mathcal{N} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$; neighbourhood sampling function $SAMPLE : 2^{\mathcal{V}} \rightarrow \mathcal{V}$

Outputs: Vector representations for each node in matrix $\mathbf{Z} \in \mathbb{R}^{N \times D}$

1 Initialise walk-set $\mathcal{W}$ to empty;
2 **for** $i = 1, .., r$ **do**
3 **for** $u \in \mathcal{V}$ **do**
4 Set $w_0 \leftarrow u$;
5 Initialise walk $\mathbf{w} = \{w_0\}$;
6 **for** $l = 0, .., L - 1$ **do**
7 $w_{l+1} \leftarrow SAMPLE(\mathcal{N}(w_l), \pi)$;
8 $\mathbf{w} \leftarrow APPEND(\mathbf{w}, w_{l+1})$;
9 **end**
10 $\mathcal{W} \leftarrow APPEND(\mathcal{W}, \mathbf{w})$;
11 **end**
12 **end**
13 Initialise random $\mathbf{Z}$;
14 **for** $\boldsymbol{w} \in \mathcal{W}$ **do**
15 **for** $w_j \in \boldsymbol{w}$ **do**
16 **for** $u \in \{w_{j-k}, \ldots, w_{j+k}\}$ **do**
17 $J(\mathbf{Z}) \leftarrow -\log \mathbb{P}(u|\mathbf{z}_{w_j})$;
18 $\mathbf{Z} \leftarrow \mathbf{Z} - \alpha \cdot \frac{\partial J}{\partial \mathbf{Z}}$;
19 **end**
20 **end**
21 **end**
22 **Return:** $\mathbf{Z} \in \mathbb{R}^{N \times D}$

2.2.2.2 Random Walk Architectures

DeepWalk and node2vec use different "$SAMPLE$" functions (Algorithm2 , Line 7) to generate their random walks, both with the aim of encapsulating the local structure and global context for each node in the graph $\mathcal{G}$.

DeepWalk The DeepWalk "$SAMPLE$" function picks the next node to visit in the walk by uniformly selecting from the current node's neighbourhood $\mathcal{N}$. This uniform distribution π, on neighbourhood $\mathcal{N}$, is proportional to the edge weights in the adjacency matrix $\mathbf{A}$. An example of such a walk is shown in Figure 2.5.

node2vec node2vec is a more advanced version of DeepWalk. One of DeepWalk's limitations is that one has no control over the style of the random walks, so one cannot tailor the way it uncovers the network structure. node2vec overcomes this by using a flexible-biased random

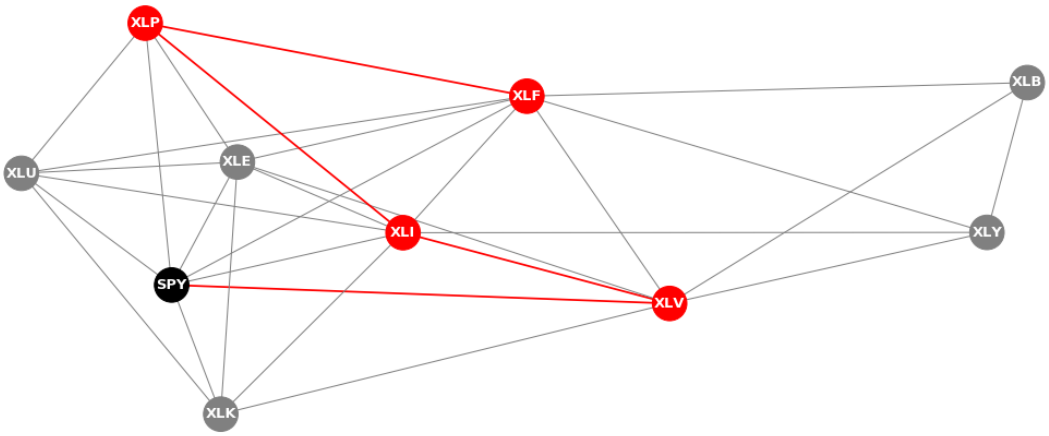


Figure 2.5: DeepWalk network exploration via uniform random walk on sample market network, starting from instrument "SPY".

walk procedure that can explore neighbourhoods with Breadth First Sampling (BFS) and Depth First Sampling (DFS) by adding extra weight parameters to each neighbouring node [11]. These two different search styles are demonstrated in Figure 2.6.

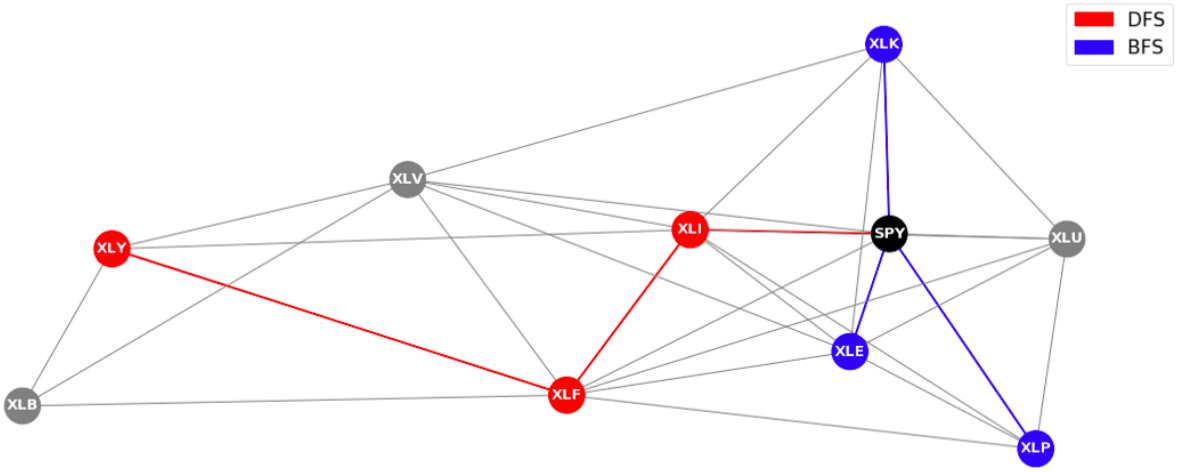


Figure 2.6: node2vec network exploration with DFS and BFS search options on sample market network starting from instrument "SPY".

The bias parameters are dependant on the walk's previous positions. Consider a random walk on $\mathcal{G}(\mathcal{V}, \mathcal{E})$ that just traversed edge (t, v) and is now at node v, as illustrated in Figure 2.7. The walk needs to decide which node to visit next, so it evaluates the transition probabilities $\pi_{v,x}$ on edges (v, x) leaving from v.

We set the un-normalized transition probability to:

$$\pi_{v,x} = \alpha_{p,q}(v,x) \cdot \mathbf{A}_{vx} \ , \tag{2.11}$$

where $\mathbf{A}_{vx}$ is the unweighted transition probability between the two nodes, and

$$\alpha_{p,q}(v,x) = \begin{cases} \frac{1}{p} & \text{if } x = t \\ 1 & \text{if } x \neq t \text{ and } (t,v) \in \mathcal{E} \\ \frac{1}{q} & \text{if } x \neq t \text{ and } (t,v) \notin \mathcal{E} \end{cases} \ . \tag{2.12}$$

A larger parameter p and smaller parameter q, leads to a lower probability of returning to previously visited nodes, inducing a more DFS-style walk. Whereas, a smaller p but larger q induces a more BFS approach, due to a higher probability of visiting nodes that are connected to the starting node.

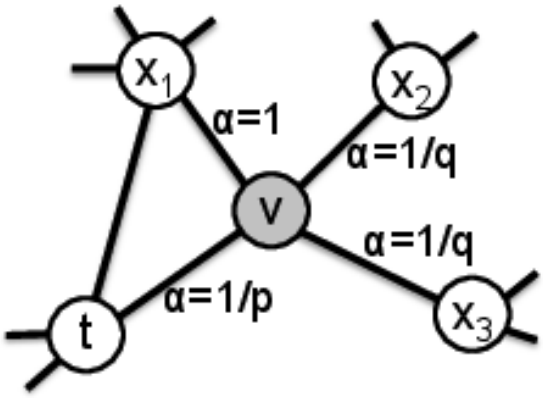


Figure 2.7: node2vec - The walk just transitioned from t to v and is now evaluating its next step out of node v, with edge labels indicating search biases $\alpha_{p,q}$ [11].

2.3 Procrustes Analysis

We can use the above embedding methods to generate an embedding of the stock market for each date. Having done this, we need a method to be able to track this temporal structure over time. The method we use in TemPr is Procrustes Analysis (PA) which is a measure of the dissimilarity between two clouds of points in a real vector space [5]. PA uses rigid transformations (linear scalings, translations and rotations) to best adjust the embedding clouds in order to minimise the element-wise distance between them.

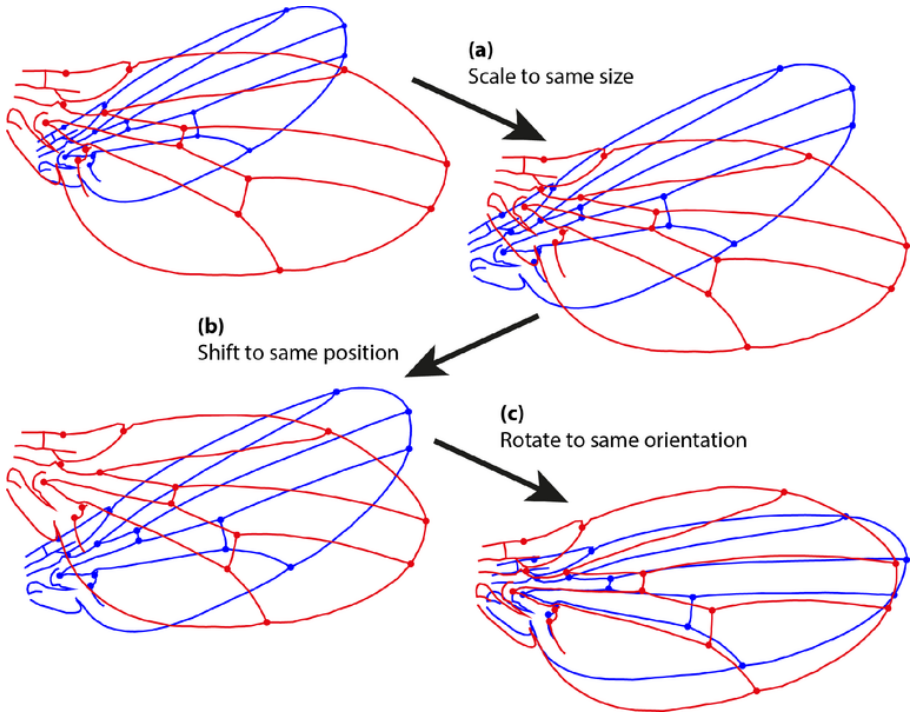


Figure 2.8: Example of how PA aligns two clusters of points using linear scalings, translations and rotations [16]

Let the embedding clouds of the market on days x and y be given by $\mathbf{Z}_x$, $\mathbf{Z}_y \in \mathbb{R}^{N\times D}$ with individual stock embeddings given by $\mathbf{x}_i$, $\mathbf{y}_i \in \mathbb{R}^D$, both indexed over $i \in \mathcal{S}$, the set of N stocks. Let $\mathbf{Z}_x$ be the *target cloud* and $\mathbf{Z}_y \in \mathbb{R}^{N\times D}$ the *test cloud*. PA then finds the *Procrustes Transformation*, P, composed of translation $\mathcal{T} \in \mathbb{R}^D$, scaling $s \in \mathbb{R}$ and rotation $\mathbf{R} \in \mathbb{R}^{D\times D}$ that minimise the total L_2 distance between points $\mathbf{x}_i$ in the target cloud, and the corresponding points $\mathbf{u}_i := P(\mathbf{y}_i) = s \cdot \mathbf{R}\mathbf{y}_i + \mathcal{T}$ in the transformed test cloud [5]. Thus, the PA dissimilarity between embeddings $\mathbf{Z}_x$ and $\mathbf{Z}_y$ is given by:

$$PA(\mathbf{Z}_x, \mathbf{Z}_y) = \min_{\mathcal{T},s,\mathbf{R}} \left\{ \sqrt{\sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathbf{u}_i||_2^2} \right\}. \tag{2.13}$$

We now proceed to derive the translation $\mathcal{T}$, scaling s and rotation $\mathbf{R}$ that minimise the associated objective function J from Equation 2.13, where:

$$J = \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathbf{u}_i||_2^2 \, . \tag{2.14}$$

We define *cloud centroids* as $C_x = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \mathbf{x}_i$, $C_y = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \mathbf{y}_i$.

Lemma 2.3.1. *[5] Let $\{\boldsymbol{x}_i : i \in \mathcal{S}\}$ be the target cloud, $\{\mathbf{y}_i : i \in \mathcal{S}\}$ the test cloud and $\{\boldsymbol{u}_i, i \in \mathcal{S}\}$ the corresponding transformed test cloud. The translation $\mathcal{T}$ that minimises the PA dissimilarity between the target cloud and transformed cloud is given by:*

$$\mathcal{T} = \mathcal{C}_x - s \cdot \boldsymbol{R}\mathcal{C}_y \, . \tag{2.15}$$

This corresponds to aligning the centroids of the target cloud and the transformed test cloud.

Proof. The centroid of the transformed test cloud is given by:

$$\mathcal{C}_u = P(\mathcal{C}_y) = s \cdot \mathbf{R}\mathcal{C}_y + \mathcal{T} \, . \tag{2.16}$$

We can then expand our objective J, given by equation 2.14:

$$\begin{aligned}
J &= \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathbf{u}_i||_2^2 \\
&= \sum_{i \in \mathcal{S}} ||(\mathbf{x}_i - \mathcal{C}_x) - (\mathbf{u}_i - \mathcal{C}_u) + (\mathcal{C}_x - \mathcal{C}_u)||_2^2 \\
&= \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + \sum_{i \in \mathcal{S}} ||\mathbf{u}_i - \mathcal{C}_u||_2^2 - 2 \sum_{i \in \mathcal{S}} \langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{u}_i - \mathcal{C}_u \rangle + \sum_{i \in \mathcal{S}} ||\mathcal{C}_x - \mathcal{C}_u||_2^2 \\
&\quad + 2 \sum_{i \in \mathcal{S}} \langle (\mathbf{x}_i - \mathcal{C}_x) - (\mathbf{u}_i - \mathcal{C}_u), \mathcal{C}_x - \mathcal{C}_u \rangle \\
&= \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + \sum_{i \in \mathcal{S}} ||\mathbf{u}_i - \mathcal{C}_u||_2^2 - 2 \sum_{i \in \mathcal{S}} \langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{u}_i - \mathcal{C}_u \rangle + |\mathcal{S}| \cdot ||\mathcal{C}_x - \mathcal{C}_u||_2^2 \\
&\quad + 2 \Big\langle \sum_{i \in \mathcal{S}} (\mathbf{x}_i - \mathcal{C}_x) - \sum_{i \in \mathcal{S}} (\mathbf{u}_i - \mathcal{C}_u), \mathcal{C}_x - \mathcal{C}_u \Big\rangle \\
&= \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + \sum_{i \in \mathcal{S}} ||\mathbf{u}_i - \mathcal{C}_u||_2^2 - 2 \sum_{i \in \mathcal{S}} \langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{u}_i - \mathcal{C}_u \rangle + N \cdot ||\mathcal{C}_x - \mathcal{C}_u||_2^2 \, ,
\end{aligned}$$

where we have used that $\sum_{i \in \mathcal{S}} (\mathbf{x}_i - \mathcal{C}_x) = 0$, $\sum_{i \in \mathcal{S}} (\mathbf{u}_i - \mathcal{C}_u) = 0$, and $|\mathcal{S}| = N$. Then, we plug in our definitions $\mathbf{u}_i = s \cdot \mathbf{R}\mathbf{y}_i + \mathcal{T}$ and $\mathcal{C}_u = s \cdot \mathbf{R}\mathcal{C}_y + \mathcal{T}$:

$$\begin{aligned}
J &= \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + \sum_{i \in \mathcal{S}} ||\mathbf{u}_i - \mathcal{C}_u||_2^2 - 2 \sum_{i \in \mathcal{S}} \langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{u}_i - \mathcal{C}_u \rangle + N \cdot ||\mathcal{C}_x - \mathcal{C}_u||_2^2 \\
&= \sum_{i \in \mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + \sum_{i \in \mathcal{S}} ||s \cdot \mathbf{R}\mathbf{y}_i - s \cdot \mathbf{R}\mathcal{C}_y||_2^2 - 2 \sum_{i \in \mathcal{S}} \langle \mathbf{x}_i - \mathcal{C}_x, s \cdot \mathbf{R}\mathbf{y}_i - s \cdot \mathbf{R}\mathcal{C}_y \rangle \\
&\quad + N \cdot ||\mathcal{C}_x - (s \cdot \mathbf{R}\mathcal{C}_y + \mathcal{T})||_2^2 \, .
\end{aligned}$$

Only the final term is dependant on translation $\mathcal{T}$, which is minimised (equal to zero) when $\mathcal{T} = \mathcal{C}_x - s \cdot \mathbf{R}\mathcal{C}_y$. For the final part, note that plugging this $\mathcal{T}$ into our definition for $\mathcal{C}_u$ gives:

$$\mathcal{C}_u = s \cdot \mathbf{R}\mathcal{C}_y + \mathcal{C}_x - s \cdot \mathbf{R}\mathcal{C}_y = \mathcal{C}_x.$$

□

Having calculated the optimal translation $\mathcal{T}$, we can now derive the optimal scaling s.

Lemma 2.3.2. *[5] Let $\{\boldsymbol{x}_i : i \in \mathcal{S}\}$ be the target cloud, $\{\boldsymbol{y}_i : i \in \mathcal{S}\}$ the test cloud, and $\{\boldsymbol{u}_i, i \in \mathcal{S}\}$ the corresponding transformed test cloud. The linear scaling s that minimises the PA dissimilarity between target cloud and transformed cloud is given by:*

$$s = \frac{\sum_{i\in\mathcal{S}} \left\langle \boldsymbol{x}_i - \mathcal{C}_x, \boldsymbol{R}(\boldsymbol{y}_i - \mathcal{C}_y) \right\rangle}{\sum_{i\in\mathcal{S}} ||(\boldsymbol{y}_i - \mathcal{C}_y)||_2^2} \,. \tag{2.17}$$

Proof. Inserting the optimal translation $\mathcal{T} = \mathcal{C}_x - s \cdot \mathbf{R}\mathcal{C}_y$ into our definition for $\mathbf{u}_i$, we have that:

$$\mathbf{u}_i = s \cdot \mathbf{R}\mathbf{y}_i + \mathcal{C}_x - s \cdot \mathbf{R}\mathcal{C}_y = s \cdot \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y) + \mathcal{C}_x \,. \tag{2.18}$$

Then, the objective function becomes:

$$\begin{aligned}
J &= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathbf{u}_i||_2^2 && (2.19)\\
&= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x - s \cdot \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y)||_2^2 && (2.20)\\
&= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + s^2 \sum_{i\in\mathcal{S}} ||\mathbf{R}(\mathbf{y}_i - \mathcal{C}_y)||_2^2 - 2s \sum_{i\in\mathcal{S}} \left\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y) \right\rangle && (2.21)\\
&= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + s^2 \sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2 - 2s \sum_{i\in\mathcal{S}} \left\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y) \right\rangle \,, && (2.22)
\end{aligned}$$

where we have used that rotation-matrix $\mathbf{R}$ is a norm-preserving isometry.

The optimal value of s is the one such that $\frac{\partial J}{\partial s} = 0$. This is because the objective J is a positive quadratic in s, and thus convex, implying that the unique stationary point is the global minimiser. We calculate the derivative of J with respect to s:

$$\frac{\partial J}{\partial s} = 2s \sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2 - 2 \sum_{i\in\mathcal{S}} \left\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y) \right\rangle.$$

Solving for s such that this derivative equals zero discovers that the optimal scaling is given by:

$$s = \frac{\sum_{i\in\mathcal{S}} \left\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y) \right\rangle}{\sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2} \,. \tag{2.23}$$

□

With this, we can finally derive the optimal rotation matrix $\mathbf{R}$.

Lemma 2.3.3. *[5] [19] Let $\{\boldsymbol{x}_i : i \in \mathcal{S}\}$ be the target cloud, $\{\boldsymbol{y}_i : i \in \mathcal{S}\}$ the test cloud, and $\{\boldsymbol{u}_i, i \in \mathcal{S}\}$ the corresponding transformed test cloud, represented by matrices $\boldsymbol{Z}_x, \boldsymbol{Z}_y$, and $\boldsymbol{Z}_u \in \mathbb{R}^{N\times D}$ respectively. Having found the optimal translation $\mathcal{T}$ and scaling s, the rotation $\boldsymbol{R}$ that minimises PA dissimilarity between target cloud and transformed cloud is given by:*

$$\boldsymbol{R} = \boldsymbol{V}\boldsymbol{U}^\intercal \,, \tag{2.24}$$

where $\boldsymbol{V}$ and $\boldsymbol{U}$ are given in the single value decomposition of the matrix $\overline{\boldsymbol{Z}}_x^\intercal \overline{\boldsymbol{Z}}_y = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^\intercal$. Here $\overline{\boldsymbol{Z}}_x, \overline{\boldsymbol{Z}}_y$ represent the centred versions of matrices $\boldsymbol{Z}_x, \boldsymbol{Z}_y$.

Proof. Having found the optimal scaling s, we plug this into our objective J in equation 2.22 which becomes:

$$\begin{aligned}
J &= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + s^2 \sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2 - 2s \sum_{i\in\mathcal{S}} \big\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y)\big\rangle \\
&= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 + \frac{\Big[\sum_{i\in\mathcal{S}} \big\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y)\big\rangle\Big]^2}{\sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2} - 2\frac{\Big[\sum_{i\in\mathcal{S}} \big\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y)\big\rangle\Big]^2}{\sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2} \\
&= \sum_{i\in\mathcal{S}} ||\mathbf{x}_i - \mathcal{C}_x||_2^2 - \frac{\Big[\sum_{i\in\mathcal{S}} \big\langle \mathbf{x}_i - \mathcal{C}_x, \mathbf{R}(\mathbf{y}_i - \mathcal{C}_y)\big\rangle\Big]^2}{\sum_{i\in\mathcal{S}} ||(\mathbf{y}_i - \mathcal{C}_y)||_2^2} \,.
\end{aligned}$$

If we centre vectors $\mathbf{x}_i, \mathbf{y}_i$ by their respective centroids, $\overline{\mathbf{x}}_i = \mathbf{x}_i - \mathcal{C}_x$, $\overline{\mathbf{y}}_i = \mathbf{y}_i - \mathcal{C}_y$, the search for the rotation $\mathbf{R}$ is reduced to finding the *maximum* of the new objective $\overline{J}$ given by:

$$\overline{J} = \sum_{i\in\mathcal{S}} \langle \overline{\mathbf{x}}_i, \mathbf{R}\overline{\mathbf{y}}_i\rangle = \Big\langle \overline{\mathbf{Z}}_x, \overline{\mathbf{Z}}_y\mathbf{R}\Big\rangle = \mathrm{Tr}\big(\overline{\mathbf{Z}}_x^\intercal \overline{\mathbf{Z}}_y\mathbf{R}\big) \,. \tag{2.25}$$

Using the singular value decomposition of $\overline{\mathbf{Z}}_x^\intercal \overline{\mathbf{Z}}_y = \mathbf{U}\mathbf{D}\mathbf{V}^\intercal$ and the cyclic property of trace Tr, we have:

$$\mathrm{Tr}\big(\overline{\mathbf{Z}}_x^\intercal \overline{\mathbf{Z}}_y\mathbf{R}\big) = \mathrm{Tr}\big(\mathbf{U}\mathbf{D}\mathbf{V}^\intercal\mathbf{R}\big) = \mathrm{Tr}\big(\mathbf{D}\mathbf{V}^\intercal\mathbf{R}\mathbf{U}\big) = \mathrm{Tr}\big(\mathbf{D}\mathbf{H}\big) \,, \tag{2.26}$$

where $\mathbf{H} = \mathbf{V}^\intercal\mathbf{R}\mathbf{U} \in \mathbb{R}^{D\times D}$ is orthogonal, as it is the product of orthogonal matrices. Thus, we have:

$$\mathrm{Tr}\big(\mathbf{D}\mathbf{H}\big) = \sum_{i=1}^{D} d_i h_{ii} \,. \tag{2.27}$$

Therefore, since d_i are non-negative numbers and $\mathrm{Tr}\big(\mathbf{D}\mathbf{H}\big)$ is maximum when $h_{ii} = 1$ for $i = 1, 2 \ldots D$ (the maximal value of in an orthogonal matrix), we have that $\mathbf{H} = \mathbf{I}$ and hence, $\mathbf{V}^\intercal\mathbf{R}\mathbf{U} = \mathbf{I}$. Thus, the $\mathbf{R}$ that minimises the main objective function J is given by:

$$\mathbf{R} = \mathbf{V}\mathbf{U}^\intercal \,. \tag{2.28}$$

□

Having computed the optimal transformation and respective PA dissimilarity between all pairs of historical market embeddings, we construct a second adjacency matrix which can be interpreted as a network, with nodes representing dates. We denote this matrix by $\widehat{\mathbf{A}} \in \mathbb{R}^{T \times T}$, where T is the number of past trading dates we have available. Entry (x, y) is given by the reciprocal of the Procrustes dissimilarity between the market embeddings on dates x and y:

$$\widehat{\mathbf{A}}_{xy} = \frac{1}{\text{PA}(\mathbf{Z}_x, \mathbf{Z}_y)} \,. \tag{2.29}$$

This can be fed through a network embedding method again to give a d-dimensional mapping, $\widehat{\mathbf{Z}} \in \mathbb{R}^{T \times d}$, of the temporal changes in the market over time.

2.4 K-Nearest Neighbours

Having generated an embedding of the stock market's temporal history, where each point represents the market's state on a historical date, we use a regression variant of KNN to leverage this information. This generates a forecast for the current date by aggregating the *true forecasts* of its nearest neighbours in the embedding. KNN is built on the assumption that dates closer together in the embedding will exhibit similar behaviour and patterns in their future time series. By *true forecast* of a date, we mean the raw stock returns that were recorded over a set window of days after this date.

Say we want to generate a h-day returns forecast, for all N stocks from date T, using date embedding $\widehat{\mathbf{Z}} \in \mathbb{R}^{T \times d}$. The KNN method can be broken down into two steps:

Step 1: First, we collect the set of K dates, called KNN_T, that correspond to the K closest points to date T in the embedding. Note, we are defining distance by the Euclidean norm $||\cdot||_2$.

Step 2: Then, we take the weighted mean of the h-length future time series for each date $t \in KNN_T$. For each date $t \in KNN_T$, the forecast contribution is weighted by w_t, the inverse square of the distance between embeddings for t and T:

$$w_t = \frac{1}{||\widehat{\mathbf{Z}}_t - \widehat{\mathbf{Z}}_T||_2^2} \,. \tag{2.30}$$

Thus, for date T, our h-day time series forecast for all N stocks is given by matrix $\mathbf{S}^{(T)} \in \mathbb{R}^{N \times h}$ where:

$$\mathbf{S}^{(T)} = \frac{\sum_{t \in KNN_T} w_t \cdot \Phi[t+1 : t+h]}{\sum_{t \in KNN_T} w_t} \,. \tag{2.31}$$

Here, we define $\Phi \in \mathbb{R}^{N \times T}$ to be our multivariate time series data of N stock returns over T days, and $\Phi[t+1 : t+h] \in \mathbb{R}^{N \times h}$ their time series over the h days after T. We let $\text{fcast}_{i,T}^{(h)} \in \mathbb{R}$ denote the h-day return forecast from day T for instrument i. This is computed by summing the ith row of $\mathbf{S}^{(T)}$.

To ensure there is no forward looking bias in our forecast, we define the time series in Φ to go up until, and including, target forecast date T. Similarly, in KNN we ignore dates that occur more recently than h days before target date T, so we do not incorporate data from dates after T in our forecast.

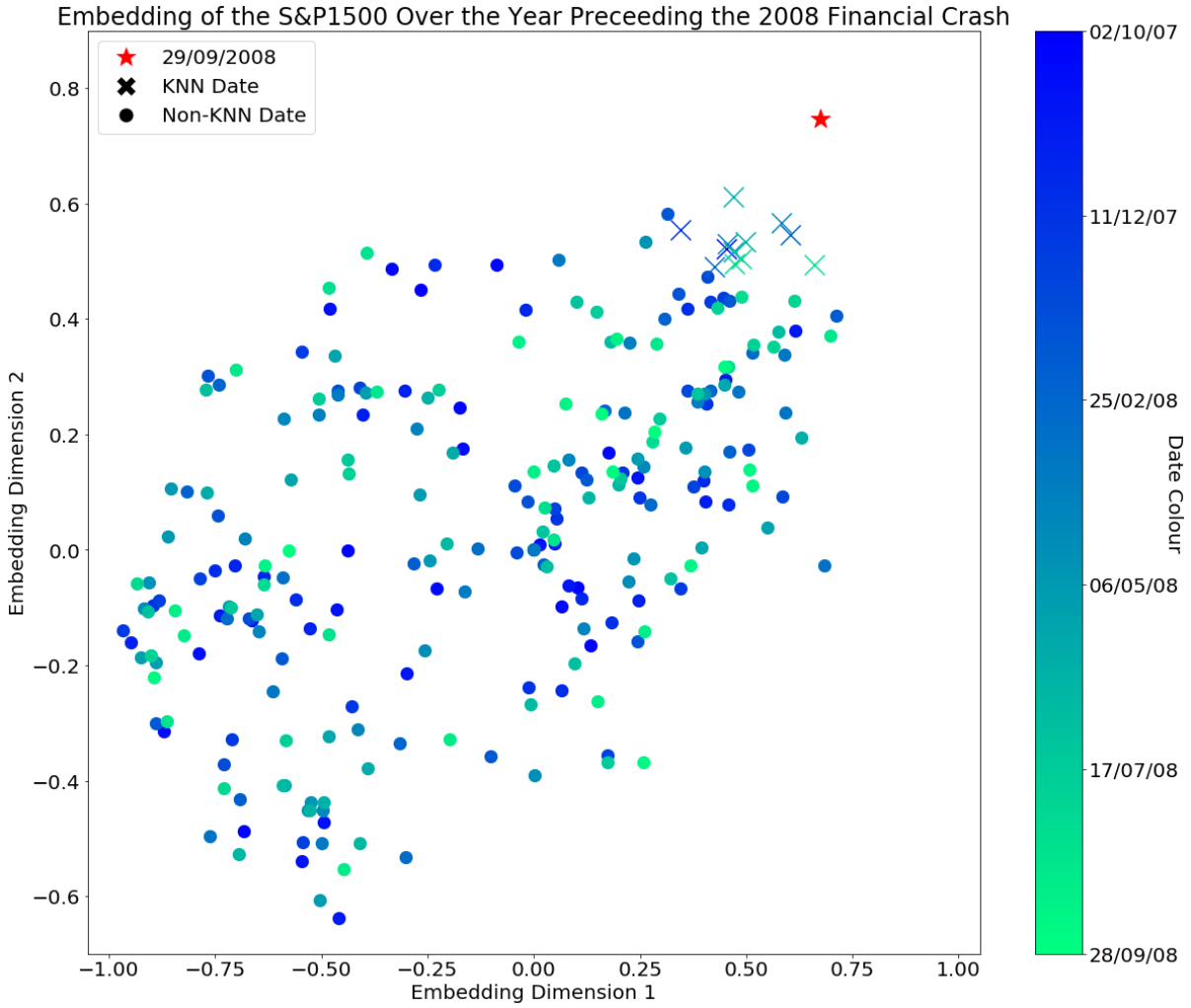


Figure 2.9: KNN on the 2D embedding $\widehat{\mathbf{Z}}$ of the S&P1500 over the year preceding the financial crash on 09/29/2008. The $K = 11$ nearest neighbours for this date have X-shaped markers. Note that, as expected, the date of the crash is an outlier in the embedding. This is due to the unprecedented market behaviour on that day substantially impacting how its 60-day tailing window correlates with other such windows.

2.5 Financial Analysis

2.5.1 Excess Market Returns

Let $\mathrm{f}_{i,t}^{(h)}$ denote the raw return of instrument i at time t, with future horizon h given by:

$$\mathrm{f}_{i,t}^{(h)} = log\frac{P_{i,t+h}}{P_{i,t}} \,, \tag{2.32}$$

where $P_{i,t}$ denotes the price of instrument i at time t. With this, we can define the *excess market return* $\bar{\mathrm{f}}_{i,t}^{(h)}$ for any instrument i given by:

$$\bar{\mathrm{f}}_{i,t}^{(h)} = \mathrm{f}_{i,t}^{(h)} - \beta \mathrm{f}_{\mathrm{SPY},t}^{(h)} \,, \tag{2.33}$$

where for simplicity we assume $\beta = 1$, and $\mathrm{f}_{\mathrm{SPY},t}^{(h)}$ denotes the raw return of ETF 'SPY'. This is the market index for S&P 500, that measures the performance of 500 large companies listed on stock exchanges in the US.

We hedge and evaluate all our forecasts $\mathrm{fcast}_{i,t}^{(h)}$ against the *excess market return* of instrument i, as it improves Sharpe Ratio (SR) performance by making the strategy more robust to systematic market fluctuations as shown by Dowd [7]. This is because instead of evaluating the raw return of an instrument, you are evaluating its performance relative to the whole market. For example, consider the case where your forecast predicts stock x will have positive return over the next 5 days, but in reality it decreased by 3% and the overall market went down 5%. In this case, without hedging you would lose your money. However, if you hedged against the excess market return, you would make money as relative to the market, stock x's value increased 2%:

$$\bar{\mathrm{f}}_x = \mathrm{f}_x - \mathrm{f}_{\mathrm{SPY}} = (-3\%) - (-5\%) = +2\%. \tag{2.34}$$

However, this can sometimes limit the gains, and even punish the forecast for being correct. For example, if the forecasts predicts x's price will increase by 2% and it does, but the total market increases by 3%, you would lose money:

$$\bar{\mathrm{f}}_x = \mathrm{f}_x - \mathrm{f}_{\mathrm{SPY}} = (+2\%) - (+3\%) = -1\%. \tag{2.35}$$

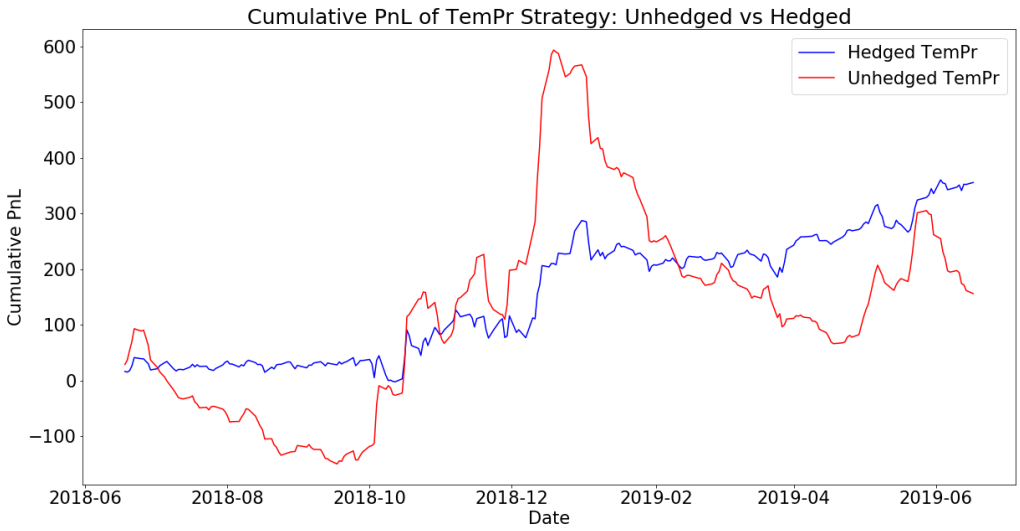


Figure 2.10: Effect on TemPr strategy of hedging against excess market return over S&P1500 2018-2019

2.5.2 Performance Metrics

In the following sections we employ three metrics to measure the success of our forecasts as described by [8].

Profit and Loss (PnL) If $\text{fcast}_{i,t}^{(h)}$ is our forecast return for instrument i, after h-days following date t, then the *PnL* of the excess market return of this forecast is given by:

$$PnL_t = \sum_{i=1}^{N} \text{sign}\big(\text{fcast}_{i,t}^{(h)}\big) \cdot \big(\text{f}_{i,t}^{(h)} - \text{f}_{\text{SPY},t}^{(h)}\big), \quad t = 1, ..., T. \tag{2.36}$$

Sharpe Ratio (SR) If we calculate the PnL of a strategy in a rolling window approach, so that PnL is a time series of length T, we can compute the annualized *Sharpe Ratio* (SR) which is defined as:

$$SR = \frac{\text{mean}(PnL)}{\text{stdev}(PnL)} \cdot \sqrt{252}\,. \tag{2.37}$$

In practice, if you compute, say a 5-day week forecast, then the multiplying factor is no longer $\sqrt{252}$, but is $\sqrt{52}$, corresponding to the 52 weeks in a year [2]. For simplicity, we keep this factor as $\sqrt{252}$ across all forecast lengths in order to easily assess the relative performance of the methods.

PnL Per Trade (PPT) We can also calculate the *PnL Per Trade* of a strategy, which is given by:

$$PPT = \frac{\sum_{t=1}^{T} PnL_t}{TN} \,, \tag{2.38}$$

where T is the number of trading days the strategy was deployed, and N is the number of instruments bet on by the strategy. For ease of visualisation, it is customary to provide the metric in basis points (bpts) where 0.0001 PPT $= 1$ bpts.

CHAPTER 3

MULTIVARIATE FORECASTING WITH TEMPR

3.1 Related Work

In order to provide context for TemPr, and asses it's predictive performance, we benchmark our technique against existing time series forecasting methods. Their capabilities span from simple univariate techniques, such as the moving average forecast, to state-of-the-art multivariate methods like LSTM.

As in Section 2.5, let $\mathrm{f}_{i,t}$ denote the raw close-to-close return of instrument i on day t given by:

$$\mathrm{f}_{i,t} = log\frac{P_{i,t}}{P_{i,t-1}} \,, \tag{3.1}$$

where $P_{i,t}$ denotes the price of instrument i at close on day t. Each method aims to generate a forecast, $\mathrm{fcast}_{i,T}^{(h)} \in \mathbb{R}$, of each instrument i and their returns for some h days following date T, given all historical values $\mathrm{f}_{i,t}$ for $t \leq T$. If the prediction for instrument i is positive, forecasting a rise in stock price, we place a long position, whereas if negative, we go short. In both cases, we hedge against market excess returns.

3.1.1 Moving Average Forecast

The moving average method predicts that the returns of a stock, on each of the next h days, will be equal to the average of returns of that stock over the last w days, for some input parameter $w \in \{1, .., T\}$. The moving average forecast return for instrument i, over h-days after date T, is given by:

$$\mathrm{fcast}_{i,T}^{(h)} = \frac{h}{w}\sum_{j=0}^{w-1} \mathrm{f}_{i,T-j} \,. \tag{3.2}$$

3.1.2 Holt's Trend Method

Holt's method [10] was devised in the 1950's and is built on the idea of exponential smoothing. It is analogous to the moving average forecast, but adds weights to the values $\mathrm{f}_{i,T-j}$, so that recent days have greater impact on the forecast. The simple exponential smoothing forecast is given by:

$$\mathrm{fcast}_{i,T}^{(h)} = \frac{\alpha h}{1-(1-\alpha)^T} \sum_{j=0}^{T-1} (1-\alpha)^j \, \mathrm{f}_{i,T-j} \, , \tag{3.3}$$

where $\alpha \in [0,1]$ is a smoothing parameter, with larger values placing even greater weight on more recent days. Holt and Brown extended this to enable the forecast to take into account a linear trend in the data, by combining exponentially smoothed forecasts for both the trend and level. The level is defined as the residual values of the time series once the trend is removed. The Holt forecast return on a particular day k days after T is given by three equations:

$$\begin{aligned}
\text{(Level Equation)} \qquad & l_T = \alpha \mathrm{f}_{i,T} + (1-\alpha)(l_{T-1} + b_{T-1}) \, , \\
\text{(Trend Equation)} \qquad & b_T = \beta(l_T - l_{T-1}) + (1-\beta) b_{T-1} \, , \\
\text{(Forecast Equation)} \qquad & \mathrm{holt}_{i,T}^{(k)} = l_T + k b_T \, ,
\end{aligned}$$

where $\alpha, \beta \in [0,1]$ are smoothing parameters. Thus, the forecast return over the h-day future window is given by:

$$\mathrm{fcast}_{i,T}^{(h)} = \sum_{k=1}^{h} \mathrm{holt}_{i,T}^{(k)} \, . \tag{3.4}$$

3.1.3 Prophet

Prophet is an automatic additive regression model that was developed by Facebook in 2017 [23]. It decomposes the returns time series $\mathrm{f}_{i,t}$ for instrument i into 4 main components:

1. $g_i(t)$ - piece-wise linear or logistic growth function of the trend, computed by detecting changes points in trends.

2. $s_i(t)$ - yearly component modelled using Fourier series.

3. $h_i(t)$ - weekly component computed using dummy variables.

4. $e_i(t)$ - user input list of 'important' dates that typically exhibit abnormal market behaviour.

Then, having learnt these functions from returns $\mathrm{f}_{i,t}$ up until date $t = T$, Prophet produces a forecast by summing each of their contributions. Hence, the forecast for instrument i, on a date k days after T, is given by:

$$\mathrm{proph}_{i,T}^{(k)} = g_i(T+k) + s_i(T+k) + h_i(T+k) + e_i(T+k) \, . \tag{3.5}$$

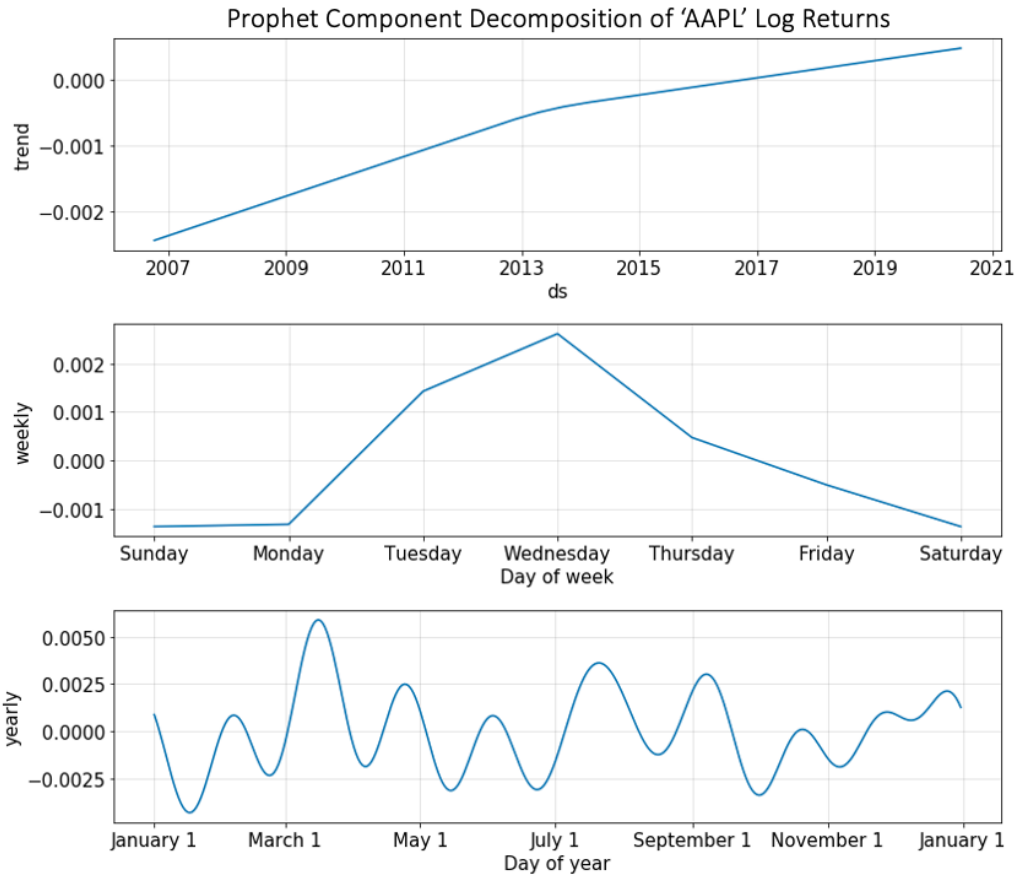


Figure 3.1: Prophet decomposition into trend $g(t)$, weekly $h(t)$ and yearly $s(t)$ components of the log-returns time series of 'AAPL' Instrument

Thus, the final Prophet forecast of the return over the h-day future window is:

$$\text{fcast}_{i,T}^{(h)} = \sum_{k=1}^{h} \text{proph}_{i,T}^{(k)} \,. \tag{3.6}$$

Intuitively, the forecast is generated in a similar fashion to Holt's method, by predicting the trend $g_i(t)$ and level $s_i(t) + h_i(t) + e_i(t)$.

3.1.4 ARIMA

While exponential smoothing models are based on a description of the level and trend in the data, Auto-Regressive Integrated Moving Average (ARIMA) describes the data using it's lags and the lagged forecast errors. For ARIMA to work, the time series must be *stationary*, which implies it has constant mean and variance over time. We achieve this by differencing the current time series $(.., \text{f}_{i,T-1}, \text{f}_{i,T})$ with itself d steps prior $(.., \text{f}_{i,T-d-1}, \text{f}_{i,T-d})$, where d is chosen to maximise the stationarity of the series. We can assume that our time series $\text{f}_{i,t}$ is stationary, because the log-return conversion process from pricing data $P_{i,t}$ differences the price time

series with $d = 1$. Hsu argues in [14] that this is sufficient to assume stationarity of time series $\mathrm{f}_{i,t}$.

The ARIMA forecast is generated by combining an auto-regressive (AR) sum of previous values in the time series with a moving average (MA) of the model's past errors, where these errors, $w_t \overset{iid}{\sim} \mathcal{N}(0, \sigma^2)$, have zero mean and identical finite variance. Thus, we iteratively define forecasts for h consecutive days after date T as follows [3]:

$$
\begin{aligned}
\text{arima}_{i,T}^{(1)} &= \delta + \alpha_1 \mathrm{f}_{i,T} + \alpha_2 \mathrm{f}_{i,T-1} + ... + \alpha_p \mathrm{f}_{i,T-p+1} + w_T + \beta_1 w_{T-1} + ... + \beta_q w_{T-q} \,, \\
\text{arima}_{i,T}^{(2)} &= \delta + \alpha_1 \text{arima}_{i,T}^{(1)} + \alpha_2 \mathrm{f}_{i,T} + ... + \alpha_p \mathrm{f}_{i,T-p+2} + \beta_1 w_T + ... + \beta_q w_{T-q+1} \,, \\
\text{arima}_{i,T}^{(3)} &= \delta + \alpha_1 \text{arima}_{i,T}^{(2)} + \alpha_2 \text{arima}_{i,T}^{(1)} + ... + \alpha_p \mathrm{f}_{i,T-p+3} + \beta_2 w_T + ... + \beta_q w_{T-q+2} \,, \\
&\vdots \\
\text{arima}_{i,T}^{(h)} &= \delta + \alpha_1 \text{arima}_{i,T}^{(h-1)} + \alpha_2 \text{arima}_{i,T}^{(h-2)} + ... + \alpha_p \mathrm{f}_{i,T-p+h} + \beta_{h-1} w_T + ... + \beta_q w_{T-q+h-1} \,,
\end{aligned}
$$

where δ, α and β are all constants determined by the model, and p, q are input parameters. In the above formulation, we assumed $h \leq p, q$. This may not always be true, in which case in the above formulation replace $\mathrm{f}_{i,T-p+h}$ by $\text{arima}_{i,T}^{(h-p)}$, and take $w_t = 0$ for all $t > T$ (as $\mathbb{E}[w_t] = 0$). Intuitively, the forecast is given by:

$$
\begin{aligned}
\text{Forecast = Constant + Linear Combination of Lags of f (up to } p \text{ lags)} \\
\text{+ Linear Combination of Lagged forecast errors (up to } q \text{ lags)} \,.
\end{aligned}
$$

We then give the final returns forecast for the whole h-day window as:

$$
\text{fcast}_{i,T}^{(h)} = \sum_{k=1}^{h} \text{arima}_{i,T}^{(k)} \,. \tag{3.7}
$$

How do we select p and q? We take plots of the following two functions to determine the optimal values for the parameters:

1. Autocorrelation Function (ACF): A measure of correlation between the time series and a lagged version of itself. For instance, at lag 5, ACF computes the correlation between $(.., \mathrm{f}_{i,T-1}, \mathrm{f}_{i,T})$ and $(.., \mathrm{f}_{i,T-6}, \mathrm{f}_{i,T-5})$.

2. Partial Autocorrelation Function (PACF): A measures of correlation between the series and its lag, after excluding the contributions from intermediate lags. For example, at lag 5, PACF computes the correlation but removes the effects already explained by lags 1 to 4.

In Figure 3.2, the blue regions on either side of 0 are 95% confidence intervals, which are used to determine parameters p and q:

- $q =$ value where the ACF chart first enters the confidence interval (here $q = 1$).
- $p =$ value where the PACF chart first enters the confidence interval (here $p = 1$)

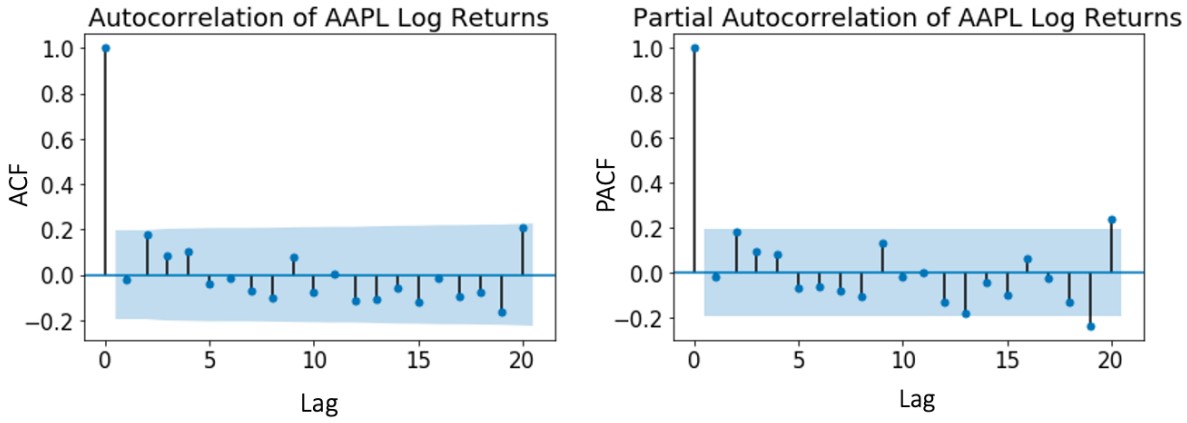


Figure 3.2: Example plots of the ACF and PACF functions for AAPL log returns over the year 2008

3.1.5 VAR

Vector Auto-Regression (VAR) can be thought of as the multivariate extension of ARIMA models. These models take the time series of returns $\{\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_T\}$ as inputs, and generate a forecast $\{\mathrm{var}_T^{(1)}, \mathrm{var}_T^{(2)}, .., \mathrm{var}_T^{(h)}\}$, by considering the linear-dependencies between the N stocks, where f_i , $\mathrm{var}_T^{(k)} \in \mathbb{R}^N$. The forecast is iteratively computed by taking the sum of the previous p values in the time series [1]:

$$\begin{aligned}
\mathrm{var}_T^{(1)} &= \epsilon + \mathbf{A}_1 \mathrm{f}_T + \mathbf{A}_2 \mathrm{f}_{T-1} + ... + \mathbf{A}_p \mathrm{f}_{T-p+1} \,, \\
\mathrm{var}_T^{(2)} &= \epsilon + \mathbf{A}_1 \mathrm{var}_T^{(1)} + \mathbf{A}_2 \mathrm{f}_T + ... + \mathbf{A}_p \mathrm{f}_{T-p+2} \,, \\
&\vdots \\
\mathrm{var}_T^{(h)} &= \epsilon + \mathbf{A}_1 \mathrm{var}_T^{(h-1)} + \mathbf{A}_2 \mathrm{var}_T^{(h-2)} + ... + \mathbf{A}_p \mathrm{f}_{T-p+h} \,,
\end{aligned}$$

where $\epsilon \in \mathbb{R}^N$, $\mathbf{A}_i \in \mathbb{R}^{N \times N}$ are constants determined by the model, and p is the lag parameter that is chosen analogously as for ARIMA. We then compute our forecast for the h-day return as:

$$\mathrm{fcast}_T^{(h)} = \sum_{k=1}^{h} \mathrm{var}_T^{(k)} \,, \tag{3.8}$$

where $\mathrm{fcast}_T^{(h)} \in \mathbb{R}^N$, which has as ith entry $\mathrm{fcast}_{i,T}^{(h)}$, corresponding to the return of stock i.

3.1.6 LSTM

Long Short-Term Memory (LSTM) is a particular type of recurrent neural network that is designed to handle sequential data, such as time series. Unlike regular feed-forward neural-networks, LSTM's have feed-back connections that allow the models to determine long-term dependencies between the sub-sequences in the time series. Given a time series of returns $\{\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_T\}$, LSTMs generates a forecast denoted by $\{\mathrm{lstm}_T^{(1)}, \mathrm{lstm}_T^{(2)}, .., \mathrm{lstm}_T^{(h)}\}$, where f_i , $\mathrm{lstm}_T^{(k)} \in \mathbb{R}^N$. They work by computing a sequence of functions, h_T and c_T, that represent the time series history. These are N-dimensional vectors and can be interpreted as memories of the LSTM, where h_T holds the short-term memory, and c_T the long-term. They are computed recursively, where h_T is a function of $\mathrm{f}_{1:T}$ and h_{T-1}, and similarly for c_T. The LSTM cell is then defined as follows [20]:

$$
\begin{aligned}
\text{(Forget Gate)} \quad & f_t = \sigma(W_f[h_{t-1}, \mathrm{f}_t] + b_f) \,, \\
\text{(Input Gate)} \quad & i_t = \sigma(W_i[h_{t-1}, \mathrm{f}_t] + b_i) \,, \\
\text{(New Information)} \quad & c'_t = \tanh(W_c[h_{t-1}, \mathrm{f}_t] + b_c) \,, \\
\text{(Cell State)} \quad & c_t = f_t \cdot c_{t-1} + i_t \cdot c'_t \,, \\
\text{(Out Gate)} \quad & o_t = \sigma(W_o[h_{t-1}, \mathrm{f}_t] + b_o) \,, \\
\text{(New State)} \quad & h_t = o_t \cdot \tanh(c_t) \,.
\end{aligned}
$$

In Figure 3.3, we show the intuition behind the LSTM cell where the system decides: what parts of the cell state c_t to forget via the forget-gate; what new information c'_t to insert via the input-gate; and, what to output to the new state h_t via the out-gate.

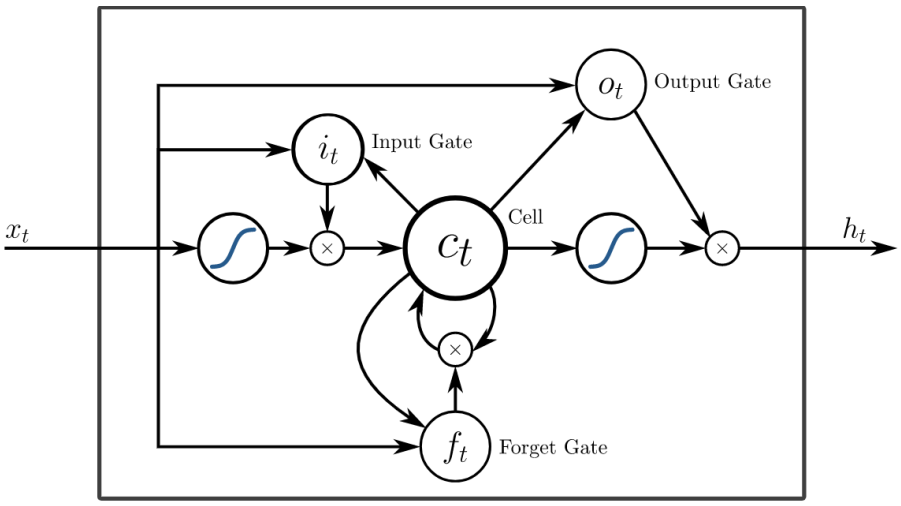


Figure 3.3: LSTM cell with forget gate [24]

This then generates forecast values by passing outputs h_t through a decoder, which is a dense layer followed by a non-linear activation function. Thus, the LSTM time series forecast

for the returns of the N stocks on date $T + k$ is given by:

$$\text{lstm}_T^{(k)} = \text{decode}(h_{T+k}) \, . \tag{3.9}$$

With this we compute our final forecast for the returns each of the N stocks over h-day future window as:

$$\text{fcast}_T^{(h)} = \sum_{k=1}^{h} \text{lstm}_T^{(k)} \, , \tag{3.10}$$

where the ith entry corresponds to the forecast returns of instrument i.

3.2 TemPr Forecasting Method

Having established the relevant theoretical framework in Chapter 2, we now proceed to sew all the segments together to describe the TemPr forecast pipeline.

On a given date T, for which we want to generate a h-day forecast, we consider the N stocks present in the S&P1500 on the every date from 01/28/2000 through to T. We assume that we have the full price and volume histories for each of these N stocks at our disposal. We then calculate the log-returns of each stock as discussed in Section 2.5, where we denote by $\text{f}_{i,t}$ the close-to-close log-return of instrument i on day t. We also compute the beta of each instrument i to the market index SPY at each time t. We calculate this by regressing their log-returns time series where the slope is given by:

$$\beta_{i,t} = \frac{\text{Cov}\big[(\text{f}_{i,1}, .., \text{f}_{i,t}), (\text{f}_{\text{SPY},1}, .., \text{f}_{\text{SPY},t})\big]}{\text{Var}\big[(\text{f}_{\text{SPY},1}, .., \text{f}_{\text{SPY},t})\big]} \, . \tag{3.11}$$

We set 04/24/2000 to be the first valid date, and date T to be the Tth valid date. This is so that each of the valid T dates have a backlog of 59 days preceding them. We define $\Phi \in \mathbb{R}^{N\times(59+T)}$ to be our set of N of excess log-returns over all historical days, where $\Phi_{ij} = \text{f}_{i,j}$ for $i = 1, ..., N, j = -58, ..., 0, 1, ..., T$. Then, we take T consecutive sub-intervals of Φ in 60-day windows, where the last date of the 1st interval is 04/24/2000, and the last date of the Tth interval corresponds to date T. Thus, we have a set of T of these $N \times 60$ matrices that we denote by $\mathcal{I}$.

Let $\phi \in \mathbb{R}^{N\times 60}$ be one such matrix in $\mathcal{I}$. We then compute the pairwise correlation between each of the N time series of ϕ and place these in adjacency matrix $\mathbf{A}_\phi \in \mathbb{R}^{N\times N}$, where we use one of the correlation measures from Section 2.1. This can be construed as a network where each node represents a stock, and the entries of $\mathbf{A}_\phi$ correspond to edge-weights in the network. Then, to every node we assign feature vectors that corresponds to the mean volume and beta of the stock it represents, over ϕ's 60-day window. We compile this information in feature matrix $\mathbf{F}_\phi \in \mathbb{R}^{N\times 2}$.

Having converted ϕ into a network with adjacency $\mathbf{A}_\phi$ and features $\mathbf{F}_\phi$, we feed this through an embedding pipeline such as GraphSAGE or node2vec. This embeds the network by outputting a 10-dimensional representation vector for each of the N stocks, described by matrix $\mathbf{Z}_\phi \in \mathbb{R}^{N\times 10}$. We then repeat this process for all $\phi \in \mathcal{I}$ so that we have T embedding matrices $\mathbf{Z}_\phi$. Each of these embeddings represents the S&P1500's temporal state over the time window it corresponds to.

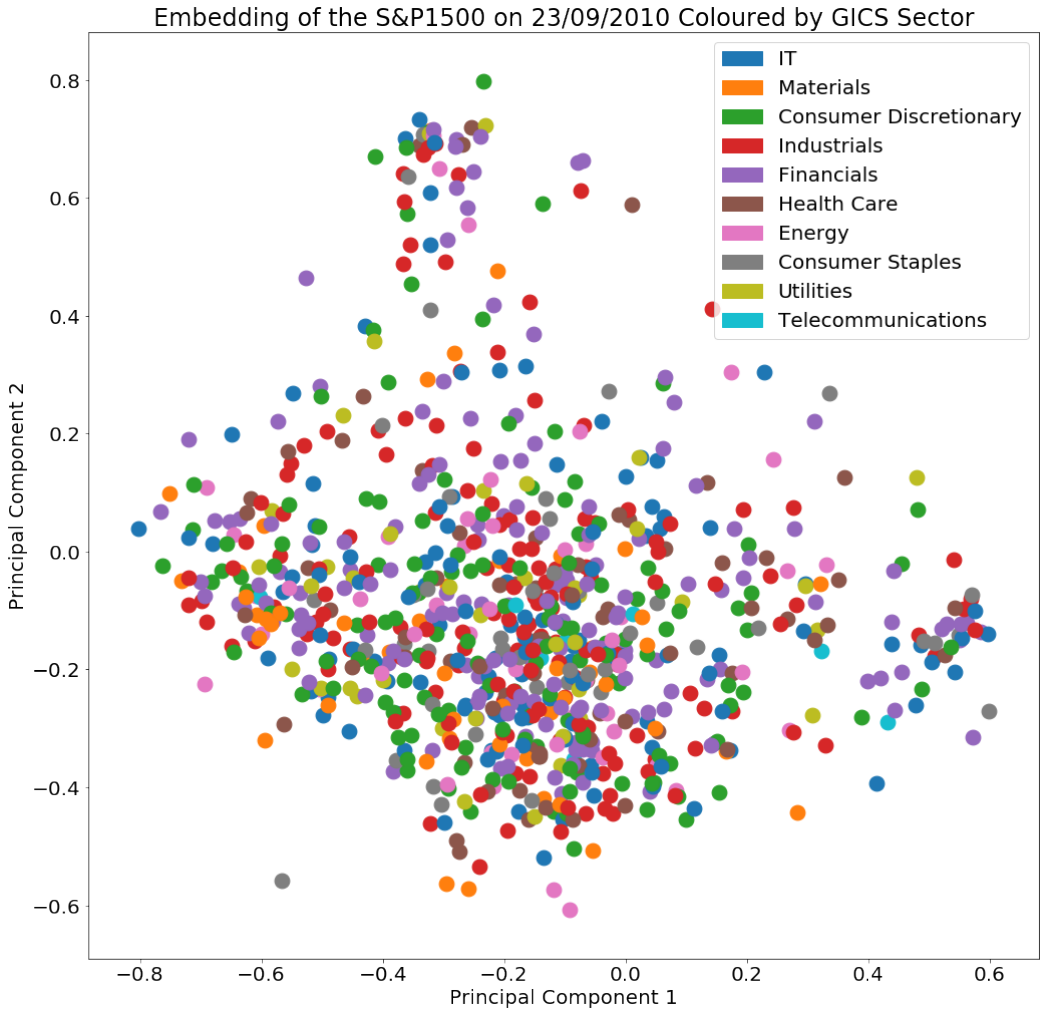


Figure 3.4: Two dimensional PCA (Principal Component Analysis) of TemPr embedding $\mathbf{Z}_\phi$ of the S&P1500 market for 22/04/2016

We compute the pairwise Procrustes dissimilarity between each of these T market embeddings. We place these in another adjacency matrix $\widehat{\mathbf{A}} \in \mathbb{R}^{T\times T}$, where entry (x, y) corresponds to the reciprocal of the Procrustes dissimilarity between market embeddings for dates x and y, as outlined in Section 2.3. Thus, we have a final network structure where now each node represents a date in the market's history. As before, we assign feature vectors that correspond to the mean volume and beta of each of the N stocks over the 60-day window preceding the

date represented by the node. We compile this information in feature matrix $\widehat{\mathbf{F}} \in \mathbb{R}^{T\times 2N}$. $\widehat{\mathbf{A}}$ and $\widehat{\mathbf{F}}$ are then fed through an embedding process, as before, to generate a 10-dimensional representation vector for each of the T dates, described by matrix $\widehat{\mathbf{Z}} \in \mathbb{R}^{T\times 10}$. This can be interpreted as a mapping of the S&P1500's temporal state over time.

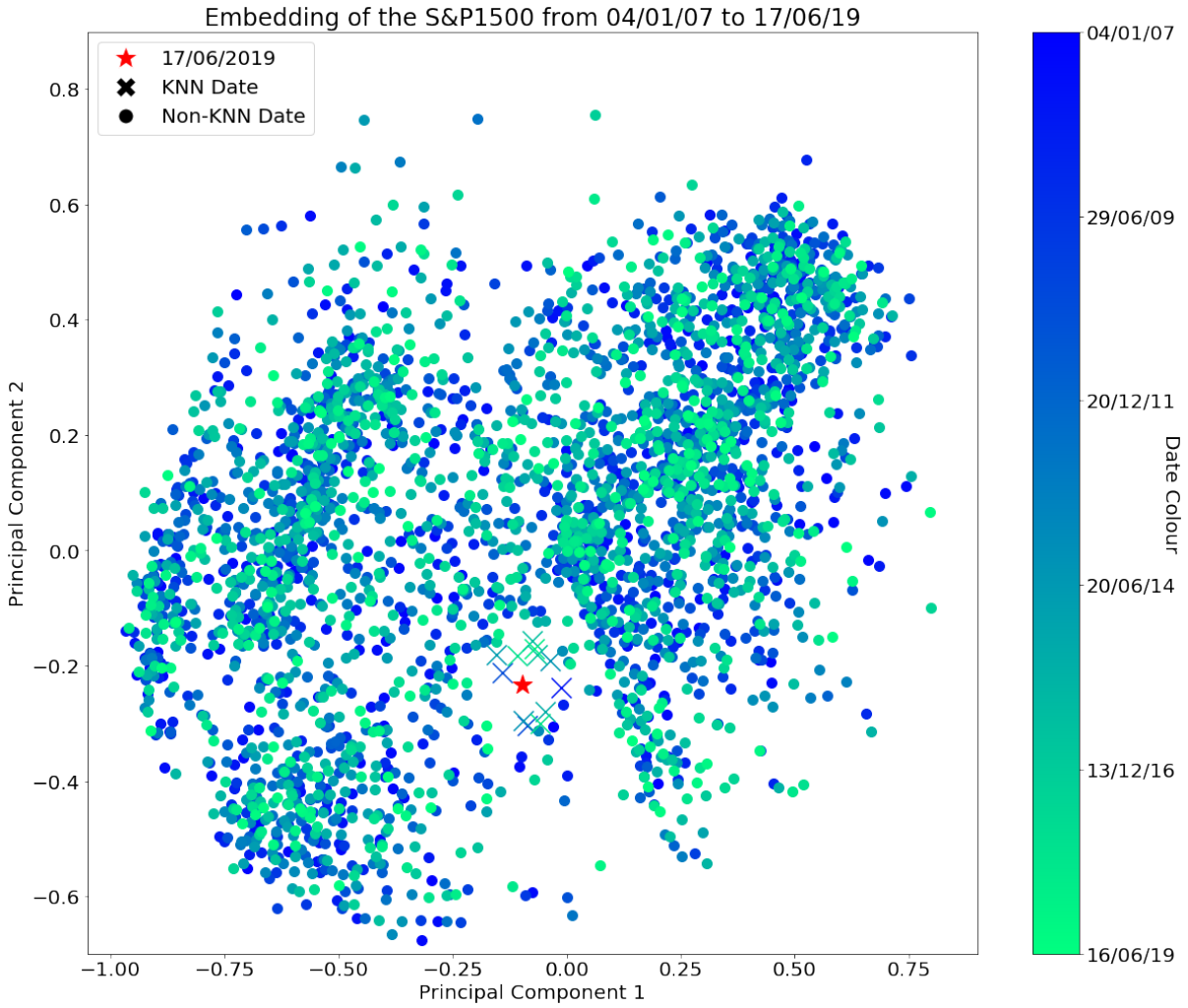


Figure 3.5: Two dimensional PCA of the TemPr S&P1500 embedding $\widehat{\mathbf{Z}}$ of the dates from 04/01/2007 to 17/06/2019, with lighter colours representing more recent dates. Highlighted in red is the date 17/06/19 and its 11 nearest neighbours are denoted by X-shaped markers.

Finally, with this mapping, a forecast from date T can be computed by applying our KNN algorithm. This algorithm outputs a forecast of the log-returns for each of the N stocks over a h-day window after date T. It starts by selecting the set KNN_T, comprised of the K closest dates to date T in the embedding $\widehat{\mathbf{Z}}$. As explained, we ignore dates that occur more recently than h days to date T, so that we do not contaminate our forecast with future values. It then sums the true returns of each stock over the h days following each date in KNN_T. The summand for each t is weighted by w_t, the reciprocal of the distance from it's date, to date T in the

embedding:

$$w_t = \frac{1}{||\widehat{\mathbf{Z}}_t - \widehat{\mathbf{Z}}_T||_2^2}\,. \tag{3.12}$$

Thus, the time series forecast $\mathbf{S}^{(T)} \in \mathbb{R}^{N\times h}$ is given by:

$$\mathbf{S}^{(T)} = \frac{\sum_{t\in KNN_T} w_t \cdot \Phi[t+1:t+h]}{\sum_{t\in KNN_T} w_t}\,. \tag{3.13}$$

For each stock, we sum its returns over this h-day forecast to give a prediction of it's h-day return. Thus, for instrument i, it's TemPr forecast return over a h-day window after date T is given by:

$$\text{fcast}_{i,T}^{(h)} = \sum_{j=1}^{h} \mathbf{S}_{ij}^{(T)}. \tag{3.14}$$

3.3 Results

In this section, we present the results used to select the optimal methods at each step in the TemPr pipeline, and benchmark its performance against existing methods. The data-set we use consists of the daily closing prices of 1029 stocks through the dates 04/01/2007 to 28/06/2019. We begin with a comparison of the three correlation techniques we can deploy to generate the initial stock-networks. We look at the induced Sharpe Ratio (SR) of the full TemPr method using each correlation measure, and compare over future window lengths $h \in \{1, 3, 5, 10\}$. For each measure, the TemPr pipeline employed GraphSAGE Mean to generate the embeddings and $k = 8$ in KNN.

Sharpe Ratios	Pearson	Distance	DTW
1 Day Close	**1.761**	1.605	0.912
3 Day Close	1.806	**2.114**	1.205
5 Day Close	2.316	**2.722**	1.107
10 Day Close	1.446	**1.633**	1.263

Table 3.1: Sharpe ratios of using different correlation methods in Tempr

In Table 3.1, we see that distance correlation performed best with the highest SRs over 3,5 & 10 day close windows. This success, compared to Pearson, may be because distance correlation can detect both linear and non-linear association between time series, making it more adept at quantifying the complex non-linear dependencies between the stock's time series. DTW can also detect non-linear dependencies by non-uniformly re-scaling the time variable. Despite this, its forecast still performed worse than Pearson, which can be attributed to the forecast is not being adjusted with this same scaling.

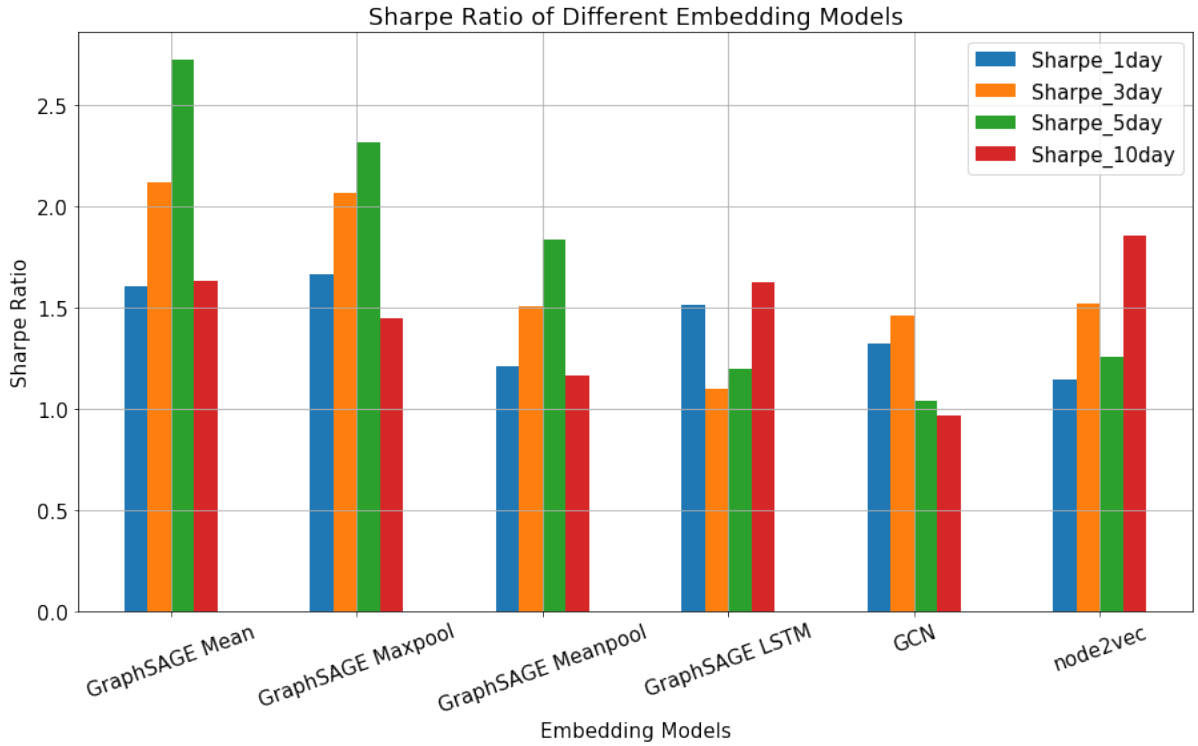


Figure 3.6: Sharpe ratios of using range of embedding models in Tempr

Similarly, Figure 3.6 displays the SRs of the TemPr strategy when we employed different embedding techniques to generate the network representations. For continuity, we used distance correlation and $k = 8$ neighbours in the TemPr strategy. We see that the best method depends on the window length, but GraphSAGE Mean performed best overall. It exhibits a distinct improvement over it's ancestoral GCN method. This can be explained by GraphSAGE's concatenation step in the aggregation process, which can be viewed as a simple form of a "skip connection" [13] between the different search depths. Hamilton et. al [12] showed this lead to significant gains in embedding performance.

Figure 3.7 displays the SRs of the TemPr strategy using a range of k-values in the KNN algorithm. Again, for continuity, we used distance correlation and GraphSAGE Mean embeddings in the TemPr pipeline. We see the overall trend is to have the highest SR for $8 \leq k \leq 12$, and the greatest SR is given by TemPr 5-day close for $k = 8$. The poor performance of smaller k values could stem from the higher variance in this forecast. For example, if we choose $k = 1$, we are relying entirely on one neighbour date to produce our forecast which may not capture the subtleties of the time series leading up to our forecast date. Similarly, if we choose k too large, we might reach too far out from our target point in the embedding and begin to incorporate dates into our forecast that do not exhibit similar patterns in their time series, highlighting a bias-variance tradeoff.

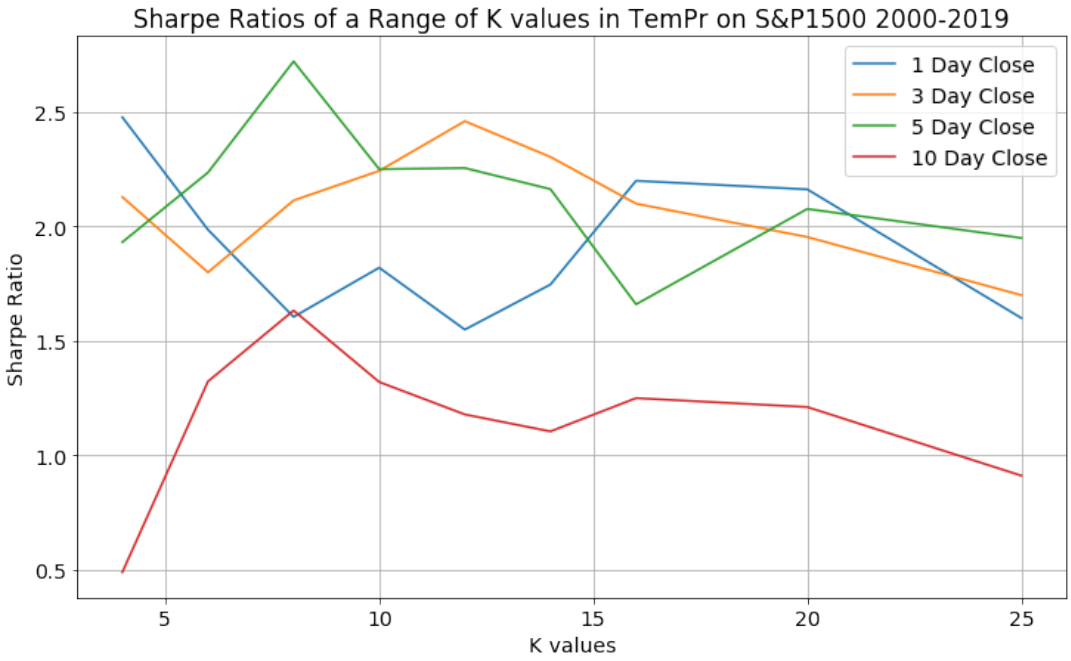


Figure 3.7: Sharpe ratios of using range of k values in Tempr

Table 3.2 displays a comparison of the optimal TemPr method (distance correlation, GraphSAGE Mean embedding and $k = 8$ neighbours) with the pre-existing time series forecasting techniques described in Section 3.1. We see that TemPr outperforms all other methods over the 3 and 5 day windows, with VAR and LSTM producing the best SR ratios over 1 and 10 days. These three methods are all multivariate techniques that consider dependencies between the stocks when generating a forecast, whereas the other techniques are purely univariate methods. This highlights how impactful it is to consider the stock prices of the market in a multivariate setting and to exploit the connections between them, as we do in TemPr.

Sharpe Ratios	TemPr	Moving Average	Holt	Prophet	ARIMA	VAR	LSTM
1 Day Close	1.605	0.190	0.933	1.022	1.258	1.323	**1.743**
3 Day Close	**2.114**	0.280	1.380	1.606	1.400	1.555	2.019
5 Day Close	**2.722**	0.347	1.401	1.548	1.371	1.766	2.434
10 Day Close	1.633	0.369	1.148	1.101	0.934	**1.687**	1.540

Table 3.2: Sharpe ratios of all forecasting strategies

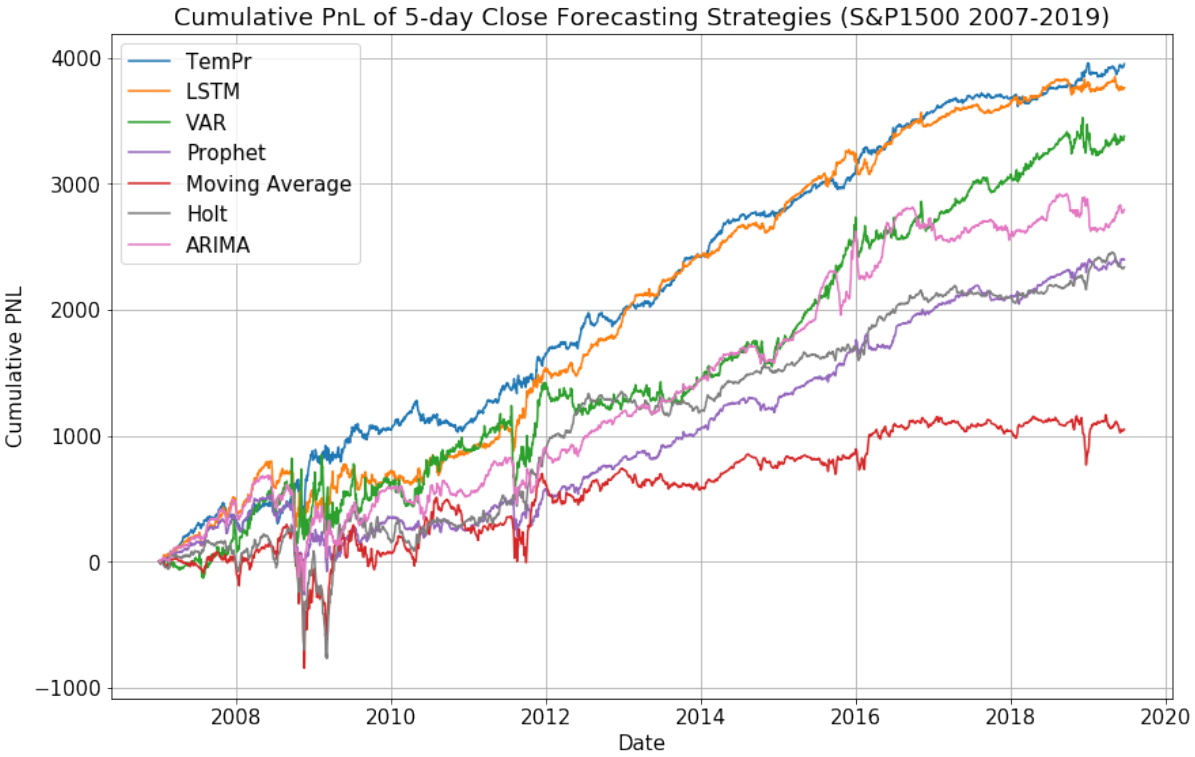


Figure 3.8: Market excess PnL of all strategies on 5-day closing positions

This hypothesis is corroborated by Figure 3.8, which shows the cumulative PnL's of each strategy from 04/01/2007 to 28/06/2019 using 5-day closing positions. We see again that TemPr, LSTM and VAR outperform the other methods. Applying further analysis to the two most successful strategies, we see that TemPr gives a slightly higher PPT of 21.3 bpts compared to LSTM's 20.4.

We also include the decile portfolios of the three most successful methods in Table 3.3, where we compared the Sharpe performance of the forecasts, as a function of their magnitudes, by trading the largest $Q\%$ of the trading signals. We see that, while TemPr is the best method over the largest two quantiles, it is also the most sensitive to change, with LSTM triumphing over the rest. This sensitivity may be attributed to the fact that volume is an input in TemPr, and so the model has greater dependency on this factor when varying trading signal magnitude.

5 Day Close	TemPr	VAR	LSTM
$Q_1 = 100$	**2.722**	1.766	2.434
$Q_2 = 75$	**2.590**	1.810	2.448
$Q_3 = 50$	2.229	1.564	**2.234**
$Q_4 = 25$	1.983	1.389	**2.105**

Table 3.3: Sharpe ratios of Tempr over quantile values

CHAPTER 4

UNIVARIATE FORECASTING WITH UNITEMPR

In this chapter, we present a univariate version of TemPr, referred to as UniTemPr.

4.1 UniTemPr Forecasting Method

Say on a given date T, we want to generate a h-day forecast for stock X. We assume we have the full history of price and volume time series for X since 01/28/2000, from which we calculate the beta. We denote the log-returns time series of X by $\mathcal{X} = (..., x_{T-1}, x_T)$. We then subset this time series into T consecutive 60-day sub-intervals, up until date T. We then calculate the pairwise distance correlation between each of these sub-intervals and place in adjacency matrix $\mathbf{A} \in \mathbb{R}^{T \times T}$, which is interpreted as a network of dates. Then, to every node-date we assign feature vectors that correspond to the mean volume and beta of X, over the 60-day window that it represents. We compile this information in feature matrix $\mathbf{F} \in \mathbb{R}^{T \times 2}$.

As with TemPr, **A** and **F** are fed through a GraphSAGE Mean embedding process that generates 10-dimensional vector representations of each date, stored in $\mathbf{Z} \in \mathbb{R}^{T \times 10}$. An analogous KNN algorithm to the one deployed in TemPr is applied to this embedding using $k = 8$. Thus, the h-day time series forecast $S_X^{(T)} \in \mathbb{R}^h$ is given by:

$$S_X^{(T)} = \frac{\sum_{t \in KNN_T} w_t \cdot \mathcal{X}[t+1 : t+h]}{\sum_{t \in KNN_T} w_t}, \tag{4.1}$$

where KNN_T is the set of 8 dates closest to T in embedding $\mathbf{Z}$, and $w_t = \frac{1}{||\mathbf{Z}_t - \mathbf{Z}_T||_2^2}$. We finally calculate the forecast return, $\text{fcast}_{X,T}^{(h)}$, by summing the h entries of $S_X^{(T)}$.

4.2 Comparing TemPr and UniTemPr to LSTM

One of the shortcomings of traditional neural-networks is that they assume all inputs are independent. This is not very practical for many tasks dealing with sequential data, such as the NLP problem of predicting the next word in a sentence, where dependence is obvious. RNNs were introduced to tackle this issue by performing the same task for every element of a sequence, where the output of the previous task is used as an input for the next. This enables the network to have some short-term memory of the input sequence. However, in this architecture, older data points contribute little to the current task so this model lacks long-term memory. Recently, LSTM's [6] have been designed to tackle this by remembering information for long periods of time, as illustrated by Figure 4.1. In the context of time series, this means that patterns in the data, across its entire history, can be used to create forecasts.

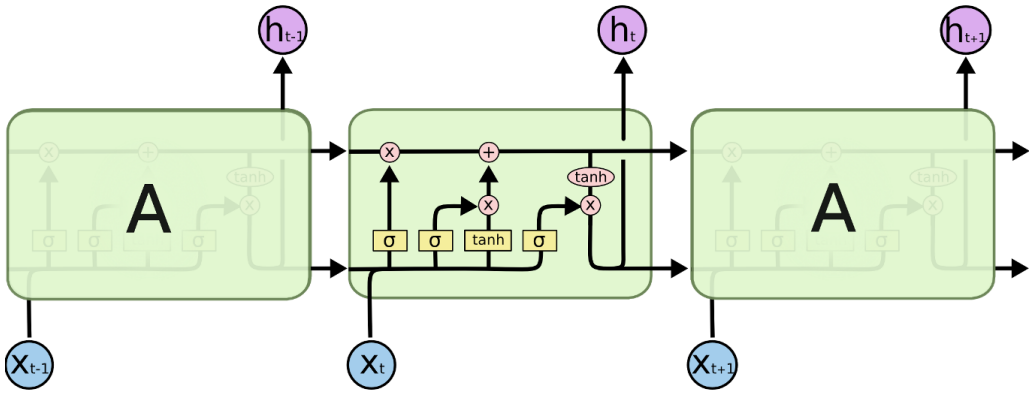


Figure 4.1: Sequence of LSTM cells which each take as input the current data point as well as an aggregation of the short and long-term memory of the data [4]

In many ways, this is comparable to the TemPr and UniTemPr techniques. In both methods, we eventually generate an embedding of all historical dates, where the proximity of dates is not chronological, but dependent on whether they exhibited similar patterns. Hence, when we produce a forecast, we are not biased against data from dates in the distant past. Thus, these methods are similar to LSTM networks in their ability to remember long-term non-linear dependencies within the data.

4.3 Results

In Table 4.1, we compare the SRs of the four most successful methods from Chapter 3 to UniTemPr. We see that UniTemPr outperforms Prophet, the best univariate method, over 1, 5 and 10-day closes. However, we observe that it under-performs compared to the multivariate techniques, further emphasising the importance of considering the stocks in a multivariate setting.

Sharpe Ratios	UniTemPr	TemPr	LSTM	VAR	Prophet
1 Day Close	1.260	1.605	**1.743**	1.323	1.022
3 Day Close	1.536	**2.114**	2.019	1.555	1.606
5 Day Close	1.769	**2.722**	2.434	1.766	1.548
10 Day Close	1.218	1.633	1.540	**1.687**	1.101

Table 4.1: Comparing the Sharpe ratio of UniTemPr to the most successful univariate and multivariate techniques

This difference is made even more stark by Figure 4.2. We see the cumulative PnL of the univariate methods (Prophet and UniTemPr) are 1000 USD less than that of the worst multivariate method (VAR). Most strikingly, the cumulative PnL of UniTemPr is roughly half that of its multivariate sibling, TemPr. This is also reflected in the PPT over 5-day closing positions where UniTemPr scores 9.1 btps compared to TemPr's 21.3. This further supports the hypothesis that stock markets should be modelled as a multivariate network.

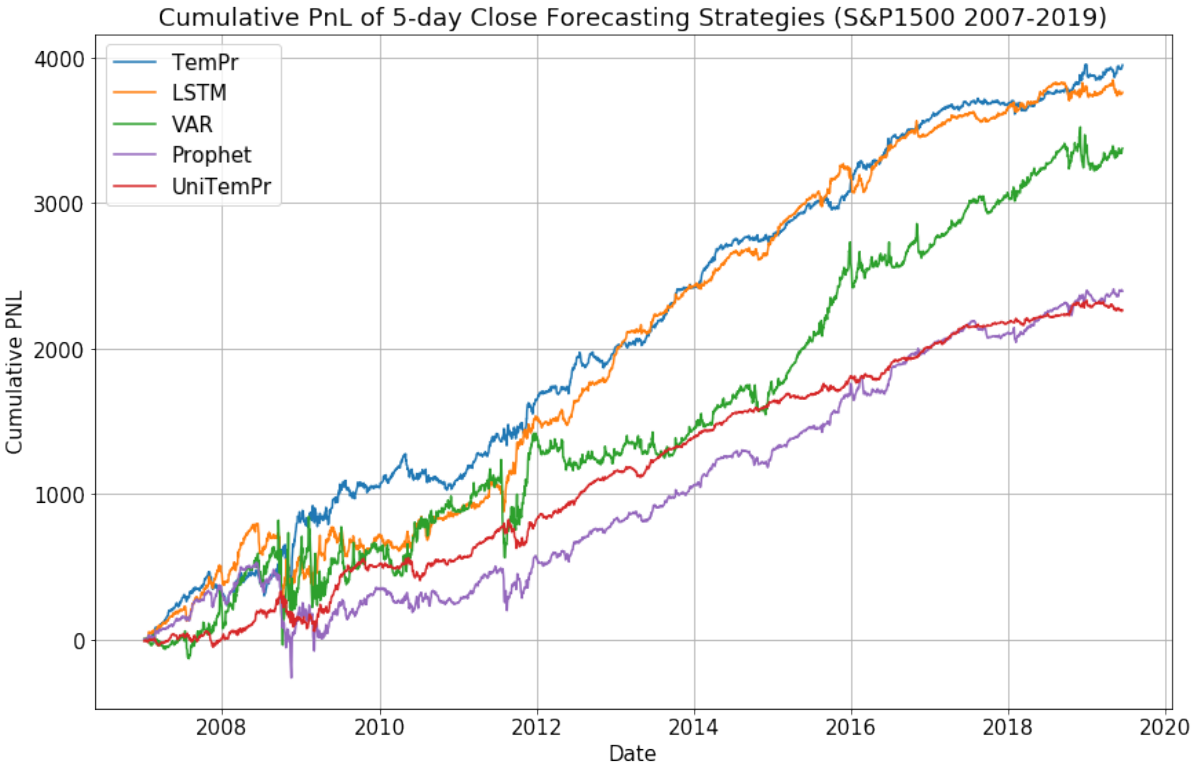


Figure 4.2: Market excess PnL of univariate and multivariate Strategies on 5-day closing positions

CHAPTER 5

DISCUSSION AND FUTURE DIRECTIONS

This dissertation has presented the multivariate forecast strategy TemPr, which uses GNNs to uncover a low-dimensional representation of the stock market on a given day, and then tracks the changes in this temporal structure. This method, unlike any other existing methods, is able to incorporate volume and beta features for each stock. Our results have demonstrated that taking this approach, and considering the market as a network of inter-dependent stocks, has the ability to outperform state-of-the-art methods like LSTM networks. We have also presented the univariate sibling of this method, UniTempr, which harnesses the power of GNNs to compare sub-intervals of a time series in order to produce a forecast. This triumphed over other univariate methods, but still fell short of the multivariate forecasting methods. This supports our hypothesis that it is crucial to consider the total temporal state of the market when producing forecasts for each stock.

Despite the success of our methods, they have their limitations, opening up areas for future investigation. One shortcoming of the TemPr methods is that they can obtain strong signals on illiquid stocks, and weak signals on liquid stocks. One interesting direction to negate this flaw could be to weight these signals by each stock's liquidity. Another possible avenue to explore could be to create a combination of TemPr and UniTemPr. This would potentially improve profits, by allowing each stock's time series to have more impact on it's own forecast, rather than being solely dependant on the entire market's temporal state. Furthermore, this may allow the incorporation of stock monitoring into the process, to dynamically close and reopen positions, rather than holding for a fixed window length.

Appendices

APPENDIX A

TIME SERIES TO NETWORK CONVERSION

```
# Function generates network from 60-day lookback stock time-series
↪  from a specified date. Later on in the pipeline we use very
↪  similar code for generating our input files for the date
↪  embedding. The only difference is we have our adjacency matrix
↪  generated using Procrustes analysis and each node has as
↪  features the volumes and beta of every stock on the date it
↪  represents.

  # Distance correlation function
  # Inputs: Arrays X,Y
  # Outputs: distance correlation of X,Y
  def distcorr(X, Y):
    X = np.atleast_1d(X)
    Y = np.atleast_1d(Y)
    if np.prod(X.shape) == len(X):
        X = X[:, None]
    if np.prod(Y.shape) == len(Y):
        Y = Y[:, None]
    X = np.atleast_2d(X)
    Y = np.atleast_2d(Y)
    n = X.shape[0]
    if Y.shape[0] != X.shape[0]:
        raise ValueError('Number of samples must match')
    a = squareform(pdist(X))
    b = squareform(pdist(Y))
    A = a - a.mean(axis=0)[None, :] - a.mean(axis=1)[:, None] +
     ↪  a.mean()
    B = b - b.mean(axis=0)[None, :] - b.mean(axis=1)[:, None] +
     ↪  b.mean()
```

```
    dcov2_xy = (A * B).sum()/float(n * n)
    dcov2_xx = (A * A).sum()/float(n * n)
    dcov2_yy = (B * B).sum()/float(n * n)
    dcor = np.sqrt(dcov2_xy)/np.sqrt(np.sqrt(dcov2_xx) *
    ↪  np.sqrt(dcov2_yy))
    return dcor


# Network Conversion Function and File Generator
# Inputs: Multivariate time series (T x N dataframe), volume and
 ↪  beta feature timeseries (both T x N dataframes), date
# Outputs: Network json file, node to ticker id map , node class
↪  map, node features, random walks from each node. These are the
↪  input files required for GraphSAGE.

def gen_gsage_inputs(timeseries, volumes, betas, date,
↪  output_directory, edge_threshold=-1):

    # Establishes inputs
    date_prefix = str(date)
    daily_ret = timeseries
    volume_df = volumes
    betas_df = betas

    ticker_list = list(adj_matrix.columns)
    total_date_list = list(daily_ret.index)
    date_index = total_date_list.index(date)

    print('Compiling Input sets..')

    ####################################
    ######### CREATING NETWORK ##########
    ####################################

    # Take 60-day subset of the N time series
    daily_ret = daily_ret.iloc[date_index-60:date_index,:]

    ### Creat Adjacency Matrix from correlation between time series
    ↪  ####

   # Creates empty ticker adjacency matrix
adj_matrix = pd.DataFrame()
for ticker in ticker_list:
    adj_matrix[ticker]=pd.Series()
    adj_matrix.loc[ticker]=pd.Series()

# Fills adjacency matrix with distance correlations
complete_ticker_list = []
for ticker1 in ticker_list:
```

```
for ticker2 in ticker_list:
    if ticker2 in complete_ticker_list:
        adj_matrix[ticker1][ticker2] =
        ↪  adj_matrix[ticker2][ticker1]
    else:
        adj_matrix[ticker1][ticker2] =
        ↪  distcorr(np.array(daily_ret[ticker1]),
        ↪  np.array(daily_ret[ticker2]))
complete_ticker_list.append(ticker1)

# Adds threshold if we want to increase sparsity of matix
adj_matrix[adj_matrix < edge_threshold] = 0
adj_matrix = adj_matrix.dropna(how='all').dropna(axis=1,
↪  how='all')

# Creating graph object G
G = nx.from_numpy_matrix(np.matrix(adj_matrix))

# Assigns each ticker id to consecutive integers and zips
↪  together into a dictionary
id_list = list(G.nodes())
ticker_id_map = dict(zip(ticker_list, id_list))


######### (1) CREATING G.JSON #########
print('Creating -G.Json')

# Creating 'test' and 'val' attributes list
int_ticker_list = G.nodes()
train_size = math.ceil(0.7*len(int_ticker_list))

test_false_list = [False]*train_size
test_true_list = [True]*(len(int_ticker_list)-train_size)

test_list = test_false_list + test_true_list
val_list = [False]*len(int_ticker_list)

# Creates list of class ids -- all set to 1 as does not affect
↪  my pipeline
class_list = [1]*len(int_ticker_list)

# Zipping dictionaries to node ids
test_dict = dict(zip(int_ticker_list, test_list))
val_dict = dict(zip(int_ticker_list, val_list))
class_dict = dict(zip(int_ticker_list, class_list))

# Adding attributes to the nodes in the graph
nx.set_node_attributes(G, "test", test_dict)
nx.set_node_attributes(G, "val", val_dict )
```

```
nx.set_node_attributes(G, "label", class_dict)

# Formatting to correct format for JSON file
ticker_returns_G = json_graph.node_link_data(G)
ticker_returns_G_graph =
 ↪  json_graph.node_link_graph(ticker_returns_G)
ticker_returns_G_graph.graph = {"name": "disjoint_union( ,  )"}
ticker_returns_G1 =
 ↪  json_graph.node_link_data(ticker_returns_G_graph)
del ticker_returns_G1['multigraph']

# Converts to correct dictionary format for JSON load
ticker_returns_G =
 ↪  json.dumps(ast.literal_eval(str(ticker_returns_G1)))

# Defines new directory for store output data files
out_dir = output_directory
if not os.path.exists(out_dir):
    os.makedirs(out_dir)

# Creates file in new directory {{ note it writes over previous
f = open(out_dir+date_prefix+"_daily_returns-G.json","w")
f.write(str(ticker_returns_G))
f.close()


######### (2) CREATING ID_MAP.JSON #########
print('Creating -id_map.Json')

# create a second map to satisfy graphsage input - this just
 ↪  maps integer to integer
id_list2 = list(range(len(id_list)))
ticker_id_map2 = dict(zip(id_list, id_list2))

# Formats to json type
ticker_returns_id_map = json.dumps(ticker_id_map2)

# Opens file and writes dictionary data to file
f = open(out_dir+date_prefix+"_daily_returns-id_map.json","w")
f.write(ticker_returns_id_map)
f.close()


######### (3) CREATING CLASS_MAP.JSON #########
print('Creating -class_map.Json')

# Converts class_dict to json format
ticker_returns_class_map = json.dumps(class_dict)
```

```
# Opens file and writes dictionary data to file
f =
 ↪  open(out_dir+date_prefix+"_daily_returns-class_map.json","w")
f.write(ticker_returns_class_map)
f.close()


######### (4) CREATING FEATS.NPY ##########
print('Creating -feats.npy')

# Takes rolling mean of volumes and beta from previous 60 days
 ↪  and retrieves from date we need
volumes_df = volumes_df.set_index('ticker')
volumes_df = volumes_df.T.rolling(window=60).mean()
volumes_feats = volumes_df.iloc[date_index,:]

betas_df = betas_df.set_index('ticker')
betas_df = betas_df.T.rolling(window=60).mean()
betas_feats = betas_df.iloc[date_index,:]

#Combines both features for each stock in one df (2 x N)
feature_df = volumes_feat.append(betas_feats)

# Creating feature file
vol_beta_features = np.array(feature_df)

# Opens file and writes dictionary data to file
np.save(out_dir+date_prefix+"_daily_returns-feats.npy",
        vol_beta_features)


######### (5) CREATING WALKS.TXT ##########
print('Creating -walks.txt')

WALK_LEN=5
N_WALKS=50

G_new = json_graph.node_link_graph(ticker_returns_G1)
nodes = int_ticker_list
pairs = []

# Generates random walks from each node on the graph
for count, node in enumerate(nodes):
    if G_new.degree(node) == 0:
        continue
    for i in range(N_WALKS):
        curr_node = node
        for j in range(WALK_LEN):
```

```
            next_node =
             ↪  random.choice(list(G_new.neighbors(curr_node)))
            # self co-occurrences are useless
            if curr_node != node:
                pairs.append((node,curr_node))
            curr_node = next_node

# writes to txt file
with open(out_dir+date_prefix+"_daily_returns-walks.txt", "w")
 ↪  as output:
    output.write('\n'.join('%s\t%s' % x for x in pairs))

print('Files compiled and stored in' + out_dir)
```

APPENDIX B

NETWORK EMBEDDING VIA GRAPHSAGE

```
# Much of this code is from
↪ https://github.com/williamleif/GraphSAGE/tree/master/graphsage
# We provide below the parts that we altered for our pipeline,
↪ including a 'load_data' function to load and pre-process the
↪ graph files and 'train' function to train the embedding. When we
↪ later generate our embedding of dates we use almost identical
↪ code to this except for minor changes to do with embedding
↪ directories.

from graphsage.minibatch import EdgeMinibatchIterator
from graphsage.neigh_samplers import UniformNeighborSampler
import graphsage.layers as layers
import graphsage.metrics as metrics
from graphsage.prediction import BipartiteEdgePredLayer
from graphsage.aggregators import MeanAggregator,
↪ MaxPoolingAggregator, MeanPoolingAggregator, SeqAggregator,
↪ GCNAggregator

############## PARAMETERS #################

# Global input parameters that need to remain constant across all
↪ embeddings
input_log_device_placement = False # """Whether to log device
↪ placement."""
input_learning_rate = 0.00001 #, 'initial learning rate.')
input_model_size    = "small" # "Can be big or small; model specific
↪ def'ns")
input_epochs = 1 #, 'number of epochs to train.')
input_dropout = 0.0 #, 'dropout rate (1 - keep probability).')
```

```
input_weight_decay = 0.0 #, 'weight for l2 loss on embedding
↪  matrix.')
input_max_degree = 100 #, 'maximum node degree.')
input_samples_1 = 25 #, 'number of samples in layer 1')
input_samples_2 = 10 #, 'number of users samples in layer 2')
input_random_context = True #, 'Whether to use random context or
↪  direct edges')
input_neg_sample_size = 20 #, 'number of negative samples')
input_n2v_test_epochs = 1 #, 'Number of new SGD epochs for n2v.')
input_identity_dim = 0 #, 'Set to positive value to use identity
↪  embedding features of that dimension. Default 0.')
#logging, saving, validation settings etc.
input_save_embeddings = True  # 'whether to save embeddings for all
↪  nodes after training')
input_base_log_dir = '.'  # 'base directory for logging and saving
↪  embeddings')
input_validate_iter = 1000  # "how often to run a validation
↪  minibatch.")
input_validate_batch_size = 256  # "how many nodes per validation
↪  sample.")
input_gpu = 1  # "which gpu to use.")
input_print_every = 500  #, "How often to print training info.")
input_max_total_steps = 10000  #, "Maximum total number of
↪  iterations")
```

```
############ MODELS.PY ###############
# models.py
class Model(object):
    def __init__(self, **kwargs):
        allowed_kwargs = {'name', 'logging', 'model_size'}
        for kwarg in kwargs.keys():
            assert kwarg in allowed_kwargs, 'Invalid keyword
            ↪  argument: ' + kwarg
        name = kwargs.get('name')
        if not name:
            name = self.__class__.__name__.lower()
        self.name = name

        logging = kwargs.get('logging', False)
        self.logging = logging

        self.vars = {}
        self.placeholders = {}

        self.layers = []
        self.activations = []
```

```
        self.inputs = None
        self.outputs = None

        self.loss = 0
        self.accuracy = 0
        self.optimizer = None
        self.opt_op = None

    def _build(self):
        raise NotImplementedError

    def build(self):
        """ Wrapper for _build() """
        with tf.variable_scope(self.name, reuse=True):
            self._build()

        # Build sequential layer model
        self.activations.append(self.inputs)
        for layer in self.layers:
            hidden = layer(self.activations[-1])
            self.activations.append(hidden)
        self.outputs = self.activations[-1]

        # Store model variables for easy access
        variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
        ↪  scope=self.name)
        self.vars = {var.name: var for var in variables}

        # Build metrics
        self._loss()
        self._accuracy()

        self.opt_op = self.optimizer.minimize(self.loss)

    def predict(self):
        pass

    def _loss(self):
        raise NotImplementedError

    def _accuracy(self):
        raise NotImplementedError

    def save(self, sess=None):
        if not sess:
            raise AttributeError("TensorFlow session not provided.")
        saver = tf.train.Saver(self.vars)
        save_path = saver.save(sess, "tmp/%s.ckpt" % self.name)
        print("Model saved in file: %s" % save_path)
```

```
    def load(self, sess=None):
        if not sess:
            raise AttributeError("TensorFlow session not provided.")
        saver = tf.train.Saver(self.vars)
        save_path = "tmp/%s.ckpt" % self.name
        saver.restore(sess, save_path)
        print("Model restored from file: %s" % save_path)


class MLP(Model):
    """ A standard multi-layer perceptron """
    def __init__(self, placeholders, dims, categorical=True,
    ↪  **kwargs):
        super(MLP, self).__init__(**kwargs)

        self.dims = dims
        self.input_dim = dims[0]
        self.output_dim = dims[-1]
        self.placeholders = placeholders
        self.categorical = categorical

        self.inputs = placeholders['features']
        self.labels = placeholders['labels']

        self.optimizer = tf.train.AdamOptimizer(learning_rate=
        ↪  input_learning_rate)

        self.build()

    def _loss(self):
        # Weight decay loss
        for var in self.layers[0].vars.values():
            #self.loss += FLAGS.weight_decay * tf.nn.l2_loss(var)
            self.loss += input_weight_decay * tf.nn.l2_loss(var)

        # Cross entropy error
        if self.categorical:
            self.loss += metrics.masked_softmax_cross_entropy(
            ↪  self.outputs, self.placeholders['labels'],
                    self.placeholders['labels_mask'])
        # L2
        else:
            diff = self.labels - self.outputs
            self.loss += tf.reduce_sum(tf.sqrt(tf.reduce_sum(diff *
            ↪  diff, axis=1)))

    def _accuracy(self):
        if self.categorical:
```

```
            self.accuracy = metrics.masked_accuracy(self.outputs,
            ↪  self.placeholders['labels'],
                    self.placeholders['labels_mask'])

    def _build(self):
        self.layers.append(layers.Dense(input_dim= self.input_dim,
                                        output_dim=self.dims[1],
                                        act=tf.nn.relu,
                                dropout=self.placeholders['dropout'],
                                        sparse_inputs=False,
                                        logging=self.logging))

        self.layers.append(layers.Dense(input_dim= self.dims[1],
                                        output_dim=self.output_dim,
                                        act=lambda x: x,
                                dropout=self.placeholders['dropout'],
                                        logging=self.logging))

    def predict(self):
        return tf.nn.softmax(self.outputs)

class GeneralizedModel(Model):
    """
    Base class for models that aren't constructed from traditional,
↪   sequential layers.
    Subclasses must set self.outputs in _build method

    (Removes the layers idiom from build method of the Model class)
    """

    def __init__(self, **kwargs):
        super(GeneralizedModel, self).__init__(**kwargs)


    def build(self):
        """ Wrapper for _build() """
        with tf.variable_scope(self.name):
            self._build()

        # Store model variables for easy access
        variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
        ↪  scope=self.name)
        self.vars = {var.name: var for var in variables}

        # Build metrics
        self._loss()
        self._accuracy()

        self.opt_op = self.optimizer.minimize(self.loss)
```

```
# SAGEInfo is a namedtuple that specifies the parameters of the
↪  recursive GraphSAGE layers
SAGEInfo = namedtuple("SAGEInfo",
    ['layer_name', # name of the layer (to get feature embedding
    ↪  etc.)
     'neigh_sampler', # callable neigh_sampler constructor
     'num_samples',
     'output_dim' # the output (i.e., hidden) dimension
    ])

class SampleAndAggregate(GeneralizedModel):
    """
    Base implementation of unsupervised GraphSAGE
    """

    def __init__(self, placeholders, features, adj, degrees,
            layer_infos, concat=True, aggregator_type="mean",
            model_size="small", identity_dim=10, **kwargs):
        '''
        Args:
            - placeholders: Stanford TensorFlow placeholder object.
            - features: Numpy array with node features.
                        NOTE: Pass a None object to train in
↪  featureless mode (identity features for nodes)!
            - adj: Numpy array with adjacency lists (padded with
↪  random re-samples)
            - degrees: Numpy array with node degrees.
            - layer_infos: List of SAGEInfo namedtuples that
↪  describe the parameters of all
                    the recursive layers. See SAGEInfo definition
↪  above.
            - concat: whether to concatenate during recursive
↪  iterations
            - aggregator_type: how to aggregate neighbor information
            - model_size: one of "small" and "big"
            - identity_dim: Set to positive int to use identity
↪  features (slow and cannot generalize, but better accuracy)
            - input_learning_rate
        '''
        super(SampleAndAggregate, self).__init__(**kwargs)
        if aggregator_type == "mean":
            self.aggregator_cls = MeanAggregator
        elif aggregator_type == "seq":
            self.aggregator_cls = SeqAggregator
        elif aggregator_type == "maxpool":
            self.aggregator_cls = MaxPoolingAggregator
        elif aggregator_type == "meanpool":
            self.aggregator_cls = MeanPoolingAggregator
```

```
        elif aggregator_type == "gcn":
            self.aggregator_cls = GCNAggregator
        else:
            raise Exception("Unknown aggregator: ",
            ↪  self.aggregator_cls)

        # get info from placeholders...
        self.inputs1 = placeholders["batch1"]
        self.inputs2 = placeholders["batch2"]
        self.model_size = model_size
        self.adj_info = adj
        identity_dim = 10
        if identity_dim > 0:
           self.embeds = tf.get_variable("node_embeddings",
            ↪  [adj.get_shape().as_list()[0], identity_dim])
        else:
           self.embeds = None
        if features is None:
            identity_dim = 10
            if identity_dim == 0:
                raise Exception("Must have a positive value for
                ↪  identity feature dimension if no input features
                ↪  given.")
            self.features = self.embeds
        else:
            self.features = tf.Variable(tf.constant(features,
            ↪  dtype=tf.float32), trainable=False)
            if not self.embeds is None:
                self.features = tf.concat([self.embeds,
                ↪  self.features], axis=1)
        self.degrees = degrees
        self.concat = concat

        self.dims = [(0 if features is None else features.shape[1])
        ↪  + identity_dim]
        self.dims.extend([layer_infos[i].output_dim for i in
        ↪  range(len(layer_infos))])
        self.batch_size = placeholders["batch_size"]
        self.placeholders = placeholders
        self.layer_infos = layer_infos

        self.optimizer = tf.train.AdamOptimizer(learning_rate=
        ↪  input_learning_rate)

        self.build()

    def sample(self, inputs, layer_infos, batch_size=None):
        """ Sample neighbors to be the supportive fields for
        ↪  multi-layer convolutions.
```

```
        Args:
            inputs: batch inputs
            batch_size: the number of inputs (different for batch
↪   inputs and negative samples).
        """

        if batch_size is None:
            batch_size = self.batch_size
        samples = [inputs]
        # size of convolution support at each layer per node
        support_size = 1
        support_sizes = [support_size]
        for k in range(len(layer_infos)):
            t = len(layer_infos) - k - 1
            support_size *= layer_infos[t].num_samples
            sampler = layer_infos[t].neigh_sampler
            node = sampler((samples[k], layer_infos[t].num_samples))
            samples.append(tf.reshape(node, [support_size *
             ↪  batch_size,]))
            support_sizes.append(support_size)
        return samples, support_sizes


    def aggregate(self, samples, input_features, dims, num_samples,
     ↪  support_sizes, batch_size=None,
            aggregators=None, name=None, concat=False,
             ↪  model_size="small"):
        """ At each layer, aggregate hidden representations of
         ↪  neighbors to compute the hidden representations
            at next layer.
        Args:
            samples: a list of samples of variable hops away for
↪   convolving at each layer of the
                network. Length is the number of layers + 1. Each is
↪   a vector of node indices.
            input_features: the input features for each sample of
↪   various hops away.
            dims: a list of dimensions of the hidden representations
↪   from the input layer to the
                final layer. Length is the number of layers + 1.
            num_samples: list of number of samples for each layer.
            support_sizes: the number of nodes to gather information
↪   from for each layer.
            batch_size: the number of inputs (different for batch
↪   inputs and negative samples).
        Returns:
            The hidden representation at the final layer for all
↪   nodes in batch
```

```
    """

    if batch_size is None:
        batch_size = self.batch_size

    # length: number of layers + 1
    hidden = [tf.nn.embedding_lookup(input_features,
     ↪  node_samples) for node_samples in samples]
    new_agg = aggregators is None
    if new_agg:
        aggregators = []
    for layer in range(len(num_samples)):
        if new_agg:
            dim_mult = 2 if concat and (layer != 0) else 1
            # aggregator at current layer
            if layer == len(num_samples) - 1:
                aggregator =
                 ↪  self.aggregator_cls(dim_mult*dims[layer],
                 ↪  dims[layer+1], act=lambda x : x,
                        dropout= self.placeholders['dropout'],
                        name=name, concat=concat,
                         ↪  model_size=model_size)
            else:
                aggregator =
                 ↪  self.aggregator_cls(dim_mult*dims[layer],
                 ↪  dims[layer+1],
                        dropout= self.placeholders['dropout'],
                        name=name, concat=concat,
                         ↪  model_size=model_size)
            aggregators.append(aggregator)
        else:
            aggregator = aggregators[layer]
        # hidden representation at current layer for all support
         ↪  nodes that are various hops away
        next_hidden = []
        # as layer increases, the number of support nodes needed
         ↪  decreases
        for hop in range(len(num_samples) - layer):
            dim_mult = 2 if concat and (layer != 0) else 1
            neigh_dims = [batch_size * support_sizes[hop],
                          num_samples[len(num_samples) - hop -
                           ↪  1],
                          dim_mult*dims[layer]]
            h = aggregator((hidden[hop],
                            tf.reshape(hidden[hop + 1],
                             ↪  neigh_dims)))
            next_hidden.append(h)
        hidden = next_hidden
    return hidden[0], aggregators
```

```
def _build(self):
    labels = tf.reshape(
            tf.cast(self.placeholders['batch2'],
             ↪  dtype=tf.int64),
            [self.batch_size, 1])
    self.neg_samples, _, _ =
     ↪  (tf.nn.fixed_unigram_candidate_sampler(
        true_classes=labels,
        num_true=1,
        num_sampled=input_neg_sample_size,
        unique=False,
        range_max=len(self.degrees),
        distortion=0.75,
        unigrams=self.degrees.tolist()))


    # perform "convolution"
    samples1, support_sizes1 = self.sample(self.inputs1,
     ↪  self.layer_infos)
    samples2, support_sizes2 = self.sample(self.inputs2,
     ↪  self.layer_infos)
    num_samples = [layer_info.num_samples for layer_info in
     ↪  self.layer_infos]
    self.outputs1, self.aggregators = self.aggregate(samples1,
     ↪  [self.features], self.dims, num_samples,
            support_sizes1, concat=self.concat,
             ↪  model_size=self.model_size)
    self.outputs2, _ = self.aggregate(samples2, [self.features],
     ↪  self.dims, num_samples,
            support_sizes2, aggregators=self.aggregators,
             ↪  concat=self.concat,
            model_size=self.model_size)

    neg_samples, neg_support_sizes =
     ↪  self.sample(self.neg_samples, self.layer_infos,
     ↪  input_neg_sample_size)

    self.neg_outputs, _ = self.aggregate(neg_samples,
     ↪  [self.features], self.dims, num_samples,
            neg_support_sizes, batch_size=input_neg_sample_size,
             ↪  aggregators=self.aggregators,
            concat=self.concat, model_size=self.model_size)

    dim_mult = 2 if self.concat else 1
    self.link_pred_layer =
     ↪  BipartiteEdgePredLayer(dim_mult*self.dims[-1],
            dim_mult*self.dims[-1], self.placeholders,
             ↪  act=tf.nn.sigmoid,
```

```
                bilinear_weights=False,
                name='edge_predict')

        self.outputs1 = tf.nn.l2_normalize(self.outputs1, 1)
        self.outputs2 = tf.nn.l2_normalize(self.outputs2, 1)
        self.neg_outputs = tf.nn.l2_normalize(self.neg_outputs, 1)

    def build(self):
        self._build()

        # TF graph management
        self._loss()
        self._accuracy()
        self.loss = self.loss / tf.cast(self.batch_size, tf.float32)
        grads_and_vars = self.optimizer.compute_gradients(self.loss)
        clipped_grads_and_vars = [(tf.clip_by_value(grad, -5.0, 5.0)
         ↪  if grad is not None else None, var)
                for grad, var in grads_and_vars]
        self.grad, _ = clipped_grads_and_vars[0]
        self.opt_op =
         ↪  self.optimizer.apply_gradients(clipped_grads_and_vars)

    def _loss(self):
        for aggregator in self.aggregators:
            for var in aggregator.vars.values():
                self.loss += input_weight_decay * tf.nn.l2_loss(var)


        self.loss += self.link_pred_layer.loss(self.outputs1,
         ↪  self.outputs2, self.neg_outputs)
        tf.summary.scalar('loss', self.loss)

    def _accuracy(self):
        # shape: [batch_size]
        aff = self.link_pred_layer.affinity(self.outputs1,
         ↪  self.outputs2)
        # shape : [batch_size x num_neg_samples]
        self.neg_aff = self.link_pred_layer.neg_cost(self.outputs1,
         ↪  self.neg_outputs)
        self.neg_aff = tf.reshape(self.neg_aff, [self.batch_size,
         ↪  input_neg_sample_size])
        _aff = tf.expand_dims(aff, axis=1)
        self.aff_all = tf.concat(axis=1, values=[self.neg_aff,
         ↪  _aff])
        size = tf.shape(self.aff_all)[1]
        _, indices_of_ranks = tf.nn.top_k(self.aff_all, k=size)
        _, self.ranks = tf.nn.top_k(-indices_of_ranks, k=size)
        self.mrr = tf.reduce_mean(tf.div(1.0, tf.cast(self.ranks[:,
         ↪  -1] + 1, tf.float32)))
```

```
        tf.summary.scalar('mrr', self.mrr)


class Node2VecModel(GeneralizedModel):
    def __init__(self, placeholders, dict_size, degrees, name=None,
                 nodevec_dim=50, lr=0.001, **kwargs):
        """ Simple version of Node2Vec/DeepWalk algorithm.

        Args:
            dict_size: the total number of nodes.
            degrees: numpy array of node degrees, ordered as in the
 ↪  data's id_map
            nodevec_dim: dimension of the vector representation of
 ↪  node.
            lr: learning rate of optimizer.
        """

        super(Node2VecModel, self).__init__(**kwargs)

        self.placeholders = placeholders
        self.degrees = degrees
        self.inputs1 = placeholders["batch1"]
        self.inputs2 = placeholders["batch2"]

        self.batch_size = placeholders['batch_size']
        self.hidden_dim = nodevec_dim

        # following the tensorflow word2vec tutorial
        self.target_embeds = tf.Variable(
                tf.random_uniform([dict_size, nodevec_dim], -1, 1),
                name="target_embeds")
        self.context_embeds = tf.Variable(
                tf.truncated_normal([dict_size, nodevec_dim],
                stddev=1.0 / math.sqrt(nodevec_dim)),
                name="context_embeds")
        self.context_bias = tf.Variable(
                tf.zeros([dict_size]),
                name="context_bias")

        self.optimizer =
         ↪  tf.train.GradientDescentOptimizer(learning_rate=lr)

        self.build()

    def _build(self):
        labels = tf.reshape(
                tf.cast(self.placeholders['batch2'],
                 ↪  dtype=tf.int64),
                [self.batch_size, 1])
```

```
        self.neg_samples, _, _ =
        ↪  (tf.nn.fixed_unigram_candidate_sampler(
            true_classes=labels,
            num_true=1,
            num_sampled=input_neg_sample_size,
            unique=True,
            range_max=len(self.degrees),
            distortion=0.75,
            unigrams=self.degrees.tolist()))

        self.outputs1 = tf.nn.embedding_lookup(self.target_embeds,
        ↪  self.inputs1)
        self.outputs2 = tf.nn.embedding_lookup(self.context_embeds,
        ↪  self.inputs2)
        self.outputs2_bias =
        ↪  tf.nn.embedding_lookup(self.context_bias, self.inputs2)
        self.neg_outputs =
        ↪  tf.nn.embedding_lookup(self.context_embeds,
        ↪  self.neg_samples)
        self.neg_outputs_bias =
        ↪  tf.nn.embedding_lookup(self.context_bias,
        ↪  self.neg_samples)

        self.link_pred_layer =
        ↪  BipartiteEdgePredLayer(self.hidden_dim, self.hidden_dim,
                self.placeholders, bilinear_weights=False)

    def build(self):
        self._build()
        # TF graph management
        self._loss()
        self._minimize()
        self._accuracy()

    def _minimize(self):
        self.opt_op = self.optimizer.minimize(self.loss)

    def _loss(self):
        aff = tf.reduce_sum(tf.multiply(self.outputs1,
        ↪  self.outputs2), 1) + self.outputs2_bias
        neg_aff = tf.matmul(self.outputs1,
        ↪  tf.transpose(self.neg_outputs)) + self.neg_outputs_bias
        true_xent = tf.nn.sigmoid_cross_entropy_with_logits(
                labels=tf.ones_like(aff), logits=aff)
        negative_xent = tf.nn.sigmoid_cross_entropy_with_logits(
                labels=tf.zeros_like(neg_aff), logits=neg_aff)
        loss = tf.reduce_sum(true_xent) +
        ↪  tf.reduce_sum(negative_xent)
        self.loss = loss / tf.cast(self.batch_size, tf.float32)
```

```
        tf.summary.scalar('loss', self.loss)

    def _accuracy(self):
        # shape: [batch_size]
        aff = self.link_pred_layer.affinity(self.outputs1,
        ↪  self.outputs2)
       # shape : [batch_size x num_neg_samples]
        self.neg_aff = self.link_pred_layer.neg_cost(self.outputs1,
        ↪  self.neg_outputs)
        self.neg_aff = tf.reshape(self.neg_aff, [self.batch_size,
        ↪  input_neg_sample_size])
        _aff = tf.expand_dims(aff, axis=1)
        self.aff_all = tf.concat(axis=1, values=[self.neg_aff,
        ↪  _aff])
        size = tf.shape(self.aff_all)[1]
        _, indices_of_ranks = tf.nn.top_k(self.aff_all, k=size)
        _, self.ranks = tf.nn.top_k(-indices_of_ranks, k=size)
        self.mrr = tf.reduce_mean(tf.div(1.0, tf.cast(self.ranks[:,
        ↪  -1] + 1, tf.float32)))
        tf.summary.scalar('mrr', self.mrr)

################ UTILS.PY ##################

def load_data(prefix, normalize=True, load_walks=False):
    G_data = json.load(open(prefix + "-G.json"))
    G = json_graph.node_link_graph(G_data)
    if isinstance(G.nodes()[0], int):
        conversion = lambda n : int(n)
    else:
        conversion = lambda n : n

    if os.path.exists(prefix + "-feats.npy"):
        feats = np.load(prefix + "-feats.npy")
        feats = feats.reshape(-1, 1)
    else:
        print("No features present.. Only identity features will be
        ↪  used.")
        feats = None
    id_map = json.load(open(prefix + "-id_map.json"))
    id_map = {conversion(k):int(v) for k,v in id_map.items()}
    walks = []
    class_map = json.load(open(prefix + "-class_map.json"))
    if isinstance(list(class_map.values())[0], list):
        lab_conversion = lambda n : n
    else:
        lab_conversion = lambda n : int(n)

    class_map = {conversion(k):lab_conversion(v) for k,v in
    ↪  class_map.items()}
```

```
    ## Remove all nodes that do not have val/test annotations
    ## (necessary because of networkx weirdness with the Reddit
    ↪  data)
    broken_count = 0
    for node in G.nodes():
        if not 'val' in G.node[node] or not 'test' in G.node[node]:
            G.remove_node(node)
            broken_count += 1
    print("Removed {:d} nodes that lacked proper annotations due to
    ↪  networkx versioning issues".format(broken_count))

    ## Make sure the graph has edge train_removed annotations
    ## (some datasets might already have this..)
    print("Loaded data.. now preprocessing..")
    for edge in G.edges():
        if (G.node[edge[0]]['val'] or G.node[edge[1]]['val'] or
            G.node[edge[0]]['test'] or G.node[edge[1]]['test']):
            G[edge[0]][edge[1]]['train_removed'] = True
        else:
            G[edge[0]][edge[1]]['train_removed'] = False

    if normalize and not feats is None:
        from sklearn.preprocessing import StandardScaler
        train_ids = np.array([id_map[n] for n in G.nodes() if not
        ↪  G.node[n]['val'] and not G.node[n]['test']])
        train_feats = feats[train_ids]
        scaler = StandardScaler()
        scaler.fit(train_feats)
        feats = scaler.transform(feats)

    if load_walks:
        with open(prefix + "-walks.txt") as fp:
            for line in fp:
                walks.append(map(conversion, line.split()))

    return G, feats, id_map, walks, class_map

#random walks variables
WALK_LEN=5
N_WALKS=50

def run_random_walks(G, nodes, num_walks=N_WALKS):
    pairs = []
    for count, node in enumerate(nodes):
        if G.degree(node) == 0:
            continue
        for i in range(num_walks):
            curr_node = node
```

```
            for j in range(WALK_LEN):
                next_node = random.choice(G.neighbors(curr_node))
                # self co-occurrences are useless
                if curr_node != node:
                    pairs.append((node,curr_node))
                curr_node = next_node
        if count % 1000 == 0:
            print("Done walks for", count, "nodes")
    return pairs

################## TRAIN.PY ###################

# log_dir, save_val_embeddings and train
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ['KMP_DUPLICATE_LIB_OK']='True'

# Set random seed
seed = 123
np.random.seed(seed)
tf.set_random_seed(seed)
tf.disable_eager_execution()
GPU_MEM_FRACTION = 0.8

# Define where to place train output/embeddings
def log_dir():
    log_dir = input_base_log_dir +
    ↪  "/LIVE_2007_2019_daily_returns_embeddings"
    if not os.path.exists(log_dir):
        os.makedirs(log_dir)
    return log_dir

# Define model evaluation function
def evaluate(sess, model, minibatch_iter, size=None):
    t_test = time.time()
    feed_dict_val = minibatch_iter.val_feed_dict(size)
    outs_val = sess.run([model.loss, model.ranks, model.mrr],
                        feed_dict=feed_dict_val)
    return outs_val[0], outs_val[1], outs_val[2], (time.time() -
    ↪  t_test)

# Incrementally evaluate model
def incremental_evaluate(sess, model, minibatch_iter, size):
    t_test = time.time()
    finished = False
    val_losses = []
    val_mrrs = []
    iter_num = 0
    while not finished:
```

```
        feed_dict_val, finished, _ =
        ↪ minibatch_iter.incremental_val_feed_dict(size, iter_num)
        iter_num += 1
        outs_val = sess.run([model.loss, model.ranks, model.mrr],
                            feed_dict=feed_dict_val)
        val_losses.append(outs_val[0])
        val_mrrs.append(outs_val[2])
    return np.mean(val_losses), np.mean(val_mrrs), (time.time() -
    ↪ t_test)

# Saves output embedding
def save_val_embeddings(sess, model, minibatch_iter, size, out_dir,
↪ mod, prefix_name, embedding_dim):
    val_embeddings = []
    finished = False
    seen = set([])
    nodes = []
    iter_num = 0
    name = prefix_name.split("/")[-1]
    while not finished:
        feed_dict_val, finished, edges =
        ↪ minibatch_iter.incremental_embed_feed_dict(size,
        ↪ iter_num)
        iter_num += 1
        outs_val = sess.run([model.loss, model.mrr, model.outputs1],
                            feed_dict=feed_dict_val)
        #ONLY SAVE FOR embeds1 because of planetoid
        for i, edge in enumerate(edges):
            if not edge[0] in seen:
                val_embeddings.append(outs_val[-1][i,:])
                nodes.append(edge[0])
                seen.add(edge[0])
    if not os.path.exists(out_dir):
        os.makedirs(out_dir)
    val_embeddings = np.vstack(val_embeddings)
    np.save(out_dir + '/' + name + '_' + mod
    ↪ +'_'+str(embedding_dim)+".npy",  val_embeddings)

def construct_placeholders():
    # Define placeholders
    placeholders = {
        'batch1' : tf.placeholder(tf.int32, shape=(None),
        ↪ name='batch1'),
        'batch2' : tf.placeholder(tf.int32, shape=(None),
        ↪ name='batch2'),
        # negative samples for all nodes in the batch
        'neg_samples': tf.placeholder(tf.int32, shape=(None,),
            name='neg_sample_size'),
```

```
        'dropout': tf.placeholder_with_default(0., shape=(),
        ↪  name='dropout'),
        'batch_size' : tf.placeholder(tf.int32, name='batch_size'),
    }
    return placeholders

# Trains the embedding model
def train(train_data, prefix, test_data=None, input_batch_size=50,
↪  input_model='graphsage_mean',
         input_dim_1=5, input_dim_2=5, input_max_total_steps=1000):
    input_train_prefix = prefix
    tf.reset_default_graph()
    G = train_data[0]
    features = train_data[1]
    id_map = train_data[2]
    embed_dim = 2*input_dim_1

    if not features is None:
        # pad with dummy zero vector
        features = np.vstack([features,
        ↪  np.zeros((features.shape[1],))])

    context_pairs = train_data[3] if input_random_context else None
    placeholders = construct_placeholders()
    minibatch = EdgeMinibatchIterator(G,
            id_map,
            placeholders, batch_size=input_batch_size,
            max_degree=input_max_degree,
            num_neg_samples=input_neg_sample_size,
            context_pairs = context_pairs)
    adj_info_ph = tf.placeholder(tf.int32,
    ↪  shape=minibatch.adj.shape)
    adj_info = tf.Variable(adj_info_ph, trainable=False,
    ↪  name="adj_info")

    if input_model == 'graphsage_mean':
        # Create model
        sampler = UniformNeighborSampler(adj_info)
        layer_infos = [SAGEInfo("node", sampler, input_samples_1,
        ↪  input_dim_1),
                       SAGEInfo("node", sampler,
                       ↪  input_samples_2, input_dim_2)]

        model = SampleAndAggregate(placeholders,
                                   features,
                                   adj_info,
                                   minibatch.deg,
                                   layer_infos=layer_infos,
                                   model_size= input_model_size,
```

```
                                    identity_dim =
                                     ↪  input_model_size,
                                    logging=True)
elif input_model == 'gcn':
    # Create model
    sampler = UniformNeighborSampler(adj_info)
    layer_infos = [SAGEInfo("node", sampler, input_samples_1,
     ↪  2*input_dim_1),
                        SAGEInfo("node", sampler,
                         ↪  input_samples_2, 2*input_dim_2)]

    model = SampleAndAggregate(placeholders,
                                    features,
                                    adj_info,
                                    minibatch.deg,
                                    layer_infos=layer_infos,
                                    aggregator_type="gcn",
                                    model_size= input_model_size,
                                    identity_dim =
                                     ↪  input_identity_dim,
                                    concat=False,
                                    logging=True)

elif input_model == 'graphsage_seq':
    sampler = UniformNeighborSampler(adj_info)
    layer_infos = [SAGEInfo("node", sampler, input_samples_1,
     ↪  input_dim_1),
                        SAGEInfo("node", sampler,
                         ↪  input_samples_2, input_dim_2)]

    model = SampleAndAggregate(placeholders,
                                    features,
                                    adj_info,
                                    minibatch.deg,
                                    layer_infos=layer_infos,
                                    identity_dim =
                                     ↪  input_identity_dim,
                                    aggregator_type="seq",
                                    model_size= input_model_size,
                                    logging=True)

elif input_model == 'graphsage_maxpool':
    sampler = UniformNeighborSampler(adj_info)
    layer_infos = [SAGEInfo("node", sampler, input_samples_1,
     ↪  input_dim_1),
                        SAGEInfo("node", sampler,
                         ↪  input_samples_2, input_dim_2)]

    model = SampleAndAggregate(placeholders,
```

```
                                    features,
                                    adj_info,
                                    minibatch.deg,
                                     layer_infos=layer_infos,
                                     aggregator_type= "maxpool",
                                     model_size= input_model_size,
                                     identity_dim =
                                      ↪  input_identity_dim,
                                     logging=True)
elif input_model == 'graphsage_meanpool':
    sampler = UniformNeighborSampler(adj_info)
    layer_infos = [SAGEInfo("node", sampler, input_samples_1,
     ↪  input_dim_1),
                          SAGEInfo("node", sampler,
                           ↪  input_samples_2, input_dim_2)]

    model = SampleAndAggregate(placeholders,
                                    features,
                                    adj_info,
                                    minibatch.deg,
                                     layer_infos=layer_infos,
                                     aggregator_type= "meanpool",
                                     model_size= input_model_size,
                                     identity_dim =
                                      ↪  input_identity_dim,
                                     logging=True)

elif input_model == 'n2v':
    model = Node2VecModel(placeholders, features.shape[0],
     ↪  minibatch.deg,
                                       #2x because graphsage uses
                                        ↪  concat
                                       nodevec_dim= 2*input_dim_1,
                                       lr=input_learning_rate)
else:
    raise Exception('Error: model name unrecognized.')

config = tf.ConfigProto(log_device_placement=
 ↪  input_log_device_placement)
config.gpu_options.allow_growth = True
config.allow_soft_placement = True

# Initialize TF session
sess = tf.Session(config=config)
merged = tf.summary.merge_all()
summary_writer = tf.summary.FileWriter(log_dir(), sess.graph)

# Init variables
```

```
sess.run(tf.global_variables_initializer(),
 ↪  feed_dict={adj_info_ph: minibatch.adj})

# Train model
train_shadow_mrr = None
shadow_mrr = None

total_steps = 0
avg_time = 0.0
epoch_val_costs = []

train_adj_info = tf.assign(adj_info, minibatch.adj)
val_adj_info = tf.assign(adj_info, minibatch.test_adj)
for epoch in range(input_epochs):
    minibatch.shuffle()

    iter = 0
    print('Epoch: %04d' % (epoch + 1))
    epoch_val_costs.append(0)
    while not minibatch.end():
        # Construct feed dictionary
        feed_dict = minibatch.next_minibatch_feed_dict()
        feed_dict.update({placeholders['dropout']:
         ↪  input_dropout})

        t = time.time()
        # Training step
        outs = sess.run([merged, model.opt_op, model.loss,
         ↪  model.ranks, model.aff_all,
                model.mrr, model.outputs1], feed_dict=feed_dict)
        train_cost = outs[2]
        train_mrr = outs[5]
        if train_shadow_mrr is None:
            train_shadow_mrr = train_mrr#
        else:
            train_shadow_mrr -= (1-0.99) * (train_shadow_mrr -
             ↪  train_mrr)

        if iter % input_validate_iter == 0:
            # Validation
            sess.run(val_adj_info.op)
            val_cost, ranks, val_mrr, duration  = evaluate(sess,
             ↪  model, minibatch,
             ↪  size=input_validate_batch_size)
            sess.run(train_adj_info.op)
            epoch_val_costs[-1] += val_cost
        if shadow_mrr is None:
            shadow_mrr = val_mrr
        else:
```

```
            shadow_mrr -= (1-0.99) * (shadow_mrr - val_mrr)

        if total_steps % input_print_every == 0:
            summary_writer.add_summary(outs[0], total_steps)

        # Print results
        avg_time = (avg_time * total_steps + time.time() - t) /
        ↪  (total_steps + 1)

        if total_steps % input_print_every == 0:
            print("Iter:", '%04d' % iter,
                  "train_loss=", "{:.5f}".format(train_cost),
                  "train_mrr=", "{:.5f}".format(train_mrr),
                  "train_mrr_ema=",
                  ↪  "{:.5f}".format(train_shadow_mrr), #
                  ↪  exponential moving average
                  "val_loss=", "{:.5f}".format(val_cost),
                  "val_mrr=", "{:.5f}".format(val_mrr),
                  "val_mrr_ema=", "{:.5f}".format(shadow_mrr), #
                  ↪  exponential moving average
                  "time=", "{:.5f}".format(avg_time))

        iter += 1
        total_steps += 1

        if total_steps > input_max_total_steps:
            break

    if total_steps > input_max_total_steps:
            break
print('Iterations used= '+ str(total_steps))
print("Optimization Finished!")
if input_save_embeddings:
    sess.run(val_adj_info.op)

    save_val_embeddings(sess, model, minibatch,
    ↪  input_validate_batch_size, log_dir(), mod=input_model,
    ↪  prefix_name=input_train_prefix, embedding_dim=embed_dim)

    if input_model == "n2v":
        # stopping the gradient for the already trained nodes
        train_ids = tf.constant([[id_map[n]] for n in
        ↪  G.nodes_iter() if not G.node[n]['val'] and not
        ↪  G.node[n]['test']],
                dtype=tf.int32)
        test_ids = tf.constant([[id_map[n]] for n in
        ↪  G.nodes_iter() if G.node[n]['val'] or
        ↪  G.node[n]['test']],
                dtype=tf.int32)
```

```
update_nodes =
 ↪  tf.nn.embedding_lookup(model.context_embeds,
 ↪  tf.squeeze(test_ids))
no_update_nodes =
 ↪  tf.nn.embedding_lookup(model.context_embeds,
 ↪  tf.squeeze(train_ids))
update_nodes = tf.scatter_nd(test_ids, update_nodes,
 ↪  tf.shape(model.context_embeds))
no_update_nodes =
 ↪  tf.stop_gradient(tf.scatter_nd(train_ids,
 ↪  no_update_nodes, tf.shape(model.context_embeds)))
model.context_embeds = update_nodes + no_update_nodes
sess.run(model.context_embeds)

# run random walks
from graphsage.utils import run_random_walks
nodes = [n for n in G.nodes_iter() if G.node[n]["val"]
 ↪  or G.node[n]["test"]]
start_time = time.time()
pairs = run_random_walks(G, nodes, num_walks=50)
walk_time = time.time() - start_time

test_minibatch = EdgeMinibatchIterator(G,
    id_map,
    placeholders, batch_size=input_batch_size,
    max_degree=input_max_degree,
    num_neg_samples=input_neg_sample_size,
    context_pairs = pairs,
    n2v_retrain=True,
    fixed_n2v=True)

start_time = time.time()
print("Doing test training for n2v.")
test_steps = 0
for epoch in range(input_n2v_test_epochs):
    test_minibatch.shuffle()
    while not test_minibatch.end():
        feed_dict =
         ↪  test_minibatch.next_minibatch_feed_dict()
        feed_dict.update({placeholders['dropout']:
         ↪  input_dropout})
        outs = sess.run([model.opt_op, model.loss,
         ↪  model.ranks, model.aff_all,
            model.mrr, model.outputs1],
             ↪  feed_dict=feed_dict)
        if test_steps % input_print_every == 0:
            print("Iter:", '%04d' % test_steps,
                  "train_loss=",
                   ↪  "{:.5f}".format(outs[1]),
```

```
                    "train_mrr=",
                     ↪  "{:.5f}".format(outs[-2]))
            test_steps += 1
    train_time = time.time() - start_time
    save_val_embeddings(sess, model, minibatch,
     ↪  input_validate_batch_size, log_dir(), mod="-test",
     ↪  prefix_name=input_train_prefix)
    print("Total time: ", train_time+walk_time)
    print("Walk time: ", walk_time)
    print("Train time: ", train_time)
```

APPENDIX C

EMBEDDING ADJACENCY GENERATION USING PROCRUSTES ANALYSIS

```
# Generate TxT adjacency matrix of date embeddings

# Inputs: npy embeddings for each date prior to current date
# Outputs: TxT adjacency matrix

# Importing every embedding name
embedding_file_list =
 ↪  glob.glob('/users/Billy/Documents/Part_C/Diss/Code/notebooks/
 ↪  LIVE_2007_2019_daily_returns_embeddings/*.npy')

# Creates dictionary of embeddings assigned to their date
new_date_list = []
embedding_dict={}
for x in embedding_file_list:
    # extract date from filename and add to list
    date = x.split('/')[-1].split('_')[0]
    new_date_list.append(date)
    # assign file to embedding dictionary
    embedding_dict[date] =
     ↪  np.load('/users/Billy/Documents/Part_C/Diss/Code/
     ↪  notebooks/LIVE_2007_2019_daily_returns_embeddings/'
                  + date + '_daily_returns_graphsage_mean_10.npy')

# Creates empty adjacency matrix with dates
embedding_corr = pd.DataFrame()
for date in new_date_list:
    print("Loop 1: Adding "+date)
    embedding_corr[date]=pd.Series()
```

```
    embedding_corr.loc[date]=pd.Series()

# Fills adjacency matrix with Procrustes Similarities
count = 0
for date1 in new_date_list:
    count+=1
    print("Loop 2: Completing "+date1+'.  Count =  '+ str(count))
    for date2 in new_date_list:
        if date1 == date2:
            embedding_corr[date1][date2] = 0
        else:
            embedding_corr[date1][date2] =
             ↪  1-procrustes(embedding_dict[date1],
             ↪  embedding_dict[date2])[-1]

# Normalises Adjacency and adds self loops
adj_matrix = embedding_corr/embedding_corr.max().max()
for date in new_date_list:
    adj_matrix[date][date] = 1

# Increase matrix sparsity by introducing edge threshold
adj_matrix1[adj_matrix1<edge_threshold]=0

# Arranging columns and rows to be in chronological order
adj_matrix.columns = map(int,adj_matrix.columns)
adj_matrix = adj_matrix1[adj_matrix.columns.sort_values()]
adj_matrix.columns = map(str, adj_matrix.columns)
adj_matrix.index = list(adj_matrix)
```

APPENDIX D

KNN ON DATE EMBEDDING

```
# KNN function creates forecast for each stock using date embedding

# Inputs: target date, embedding, K
# Outputs: 10 day forecast for 20190617

def KNN_forecast(target_date ,K, embedding_dir):
    ########## LOAD Embedding ###########
    date_embedding_df = pd.DataFrame(np.load(embedding_dir)).T

    # Swapping column names (integer ids) to tickers names
    rev_id_map = { v:k for k,v in date_id_map.items()}
    date_embedding_df.columns = [rev_id_map.get(item,item)  for item
    ↪  in list(date_embedding_df.columns)]

    # Selectc date column and all columns from 10 days and more ago
    target_date_index =
    ↪  list(date_embedding_df.columns).index(target_date)
    sub_date_embedding_df =
    ↪  date_embedding_df.iloc[:,:target_date_index]

    ########## Creates df of ascending euclidean distances from
    ↪  20190617 #########
    euclid_distance_df=pd.DataFrame()
    euclid_distance_df['target_date']=pd.Series()
    euclid_distance_df['dist_from_target_date']=pd.Series()

    for date in list(sub_date_embedding_df):
        dist = np.linalg.norm(date_embedding_df[target_date] -
        ↪  date_embedding_df[date])
        euclid_distance_df=
        ↪  euclid_distance_df.append(pd.Series([date, dist], index=
        ↪  euclid_distance_df.columns), ignore_index= True)
```

```
euclid_distance_df =
 ↪  euclid_distance_df.sort_values('dist_from_target_date')

############ Makes list of KNN to 20181214 ############
knn_target_date =
 ↪  list(euclid_distance_df['target_date'].iloc[1:K+1])

########### Dictionary of nearest neighbours and their future 10
 ↪  day returns dataframes #######
neighbour_future_dict={}
for neighbour in knn_target_date:
    neighbour_index= list(daily_log_ret.index).index(neighbour)
    neighbour_future_dict[neighbour]=
     ↪  daily_log_ret.iloc[neighbour_index+1:
     ↪  neighbour_index+11,:]

####### Generates KNN 10 day forecast dataframe ######
forecast_df = daily_log_ret.iloc[:10].reset_index(drop=True)
forecast_df.iloc[:,:]=0
total_weight= ((1/euclid_distance_df['dist_from_target_date']
 ↪  .iloc[1:11])**2).sum()

for neighbour in knn_target_date:
    weight = (1/(float(euclid_distance_df
     ↪  [euclid_distance_df['target_date']==neighbour]
     ↪  ['dist_from_target_date'])))**2
    forecast_df +=  weight*
     ↪  (neighbour_future_dict[neighbour].reset_index(drop=
     ↪  True))/total_weight

############ returns forecast dataframe ########
return forecast_df
```

BIBLIOGRAPHY

[1] D. ASTERIOU and S. HALL. *Vector Autoregressive (VAR) Models and Causality Tests*, pages 333–346. Springer, 2016.

[2] E. BENHAMOU, D. SALTIEL, B. GUEZ, and N. PARIS. *Testing Sharpe ratio: luck or skill?* Available at SSRN 3391214, 2019.

[3] G. BOX, G. JENKINS, G. REINSEL, and G. LJUNG. *Time Series Analysis: Forecast and Control, 5th Edition*. Wiley, 2015.

[4] C. OLAH. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. [Online; accessed 16-March-2020].

[5] S. CAMIZ and J. DENIMAL. Procrustes analysis and stock markets. In *CS-BIGS 4(2)*, pages 93–100, 2011.

[6] G. DING and L. QIN. Study on the prediction of stock price based on the associated network model of LSTM. In *International Journal of Machine Learning and Cybernetics*, 2019.

[7] K. DOWD. Adjusting for risk: An improved sharpe ratio. In *International Review of Economics & Finance 9*, pages 209–222, 2000.

[8] W. FERSON, J. CHRISTOPHERSON, and D. CARINO. *Portfolio Performance Measurement and Benchmarking*. McGraw-Hill, 2009.

[9] D. GARCIA-TORRES and H. QUI. Applying Recurrent Neural Networks for Multivariate Time Series Forecasting of Volatile Financial Data. *Research Gate*, 2018.

[10] E. GARDNER. Exponential smoothing: The state of the art. In *Journal of Forecasting 4(2)*, pages 1–28, 1985.

[11] A. GROVER and J. LESKOVEC. node2vec: Scalable feature learning for networks. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 22*. ACM, 2016.

[12] W. HAMILTON, R. YING, and J. LESKOVEC. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30*, pages 1024–1034, 2017.

[13] K. HE, X. ZHANG, S. REN, and J. SUN. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[14] D. HSU. The behavior of stock returns: Is it stationary of evolutionary? In *The Journal of Financial and Quantitative Analysis 19(1)*, pages 11–28. Cambridge University Press, 1984.

[15] T. KIPF and M. WELLING. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[16] C. KLINGENBERG. Analyzing fluctuating asymmetry with geometric morphometrics: Concepts, methods, and applications. In *Symmetry 7(2)*, pages 843–934, 2015.

[17] Y. LI, H. CHEN, and Z. WU. Dynamic Time Warping Distance Method for Similarity Test of Multipoint Ground Motion Field. *Hindawi Publishing Corporation*, 2010.

[18] B. PEROZZI, R. AL-RFOU, and S. SKIENA. Deepwalk: Online learning of social representations. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 20*, pages 701–710. ACM, 2014.

[19] A. ROSS. Procrustes analysis. *CiteSeerX*, 2005.

[20] S. SHIH, F. SUN, and H. LEE. Temporal pattern attention for multivariate time series forecasting. In *Machine Learning, 108(8-9)*, pages 1421–1441, 2018.

[21] E. STRINGHAM, N. CUROTT, C. COYNE, and P. BOETTKE. *The Oxford Handbook of Austrian Economics*. Oxford University Press, 2015.

[22] G. SZEKELY, M. RIZZO, and N. BAKIROV. Measuring and testing dependence by correlation of distances. In *Annals of Statistics 2007 (6)*, pages 2769–2794, 2007.

[23] S. TAYLOR and B. LENTHAM. Forecasting at Scale. *PeerJ Preprints*, 2017.

[24] Wikipedia Contributors. Long short-term memory. `https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=945023662`, 2020. [Online; accessed 11-March-2020].

[25] R. ÖZCELIK. An intuitive explanation of graphsage. https://towardsdatascience.com/an-intuitive-explanation-of-graphsage-6df9437ee64f, 2019. [Online; accessed 27-February-2020].