Paper 3154-2019

Modeling with Deep Recurrent Architectures: A Case Study of Using SAS and Python for Deep Learning

Linh Le, Institute of Analytics and Data Science; Ying Xie, Department of Information Technology;
Kennesaw State University

ABSTRACT

Deep learning is attracting more and more researchers and analysts with its numerous breakthrough successes in different areas of analytics and artificial intelligence. Although being well-known for its power in data processing and manipulation, SAS® is still relatively new as a deep learning toolbox. On the other hand, Python has become "the language for deep learning" due to its flexibilities and great supports from deep learning researchers. In this paper, we present a case study of combining SAS and Python for the task of predicting next-day stock price direction with deep recurrent architectures, including vanilla Recurrent Neural Network, Long Short-Term Memory, Gated Recurrent Unit, and a novel model we proposed named Recurrent Embedding Kernel. To use the power of each framework, we preprocess data in SAS and build our deep models in Python. The whole process is unified in one framework using the SASPy module, which allows access to SAS codes in Python environment.

INTRODUCTION

Deep learning is a recent emerged field with numerous breakthrough successes in different areas of analytics and artificial intelligence. In brief, deep learning models utilize a high number of parameters stacked by layers to non-linearly map data to a feature space where decisions are made. Both the mapping and the decision function are learned from data which gives deep models great robustness and flexibility. Different deep network architectures are also designed for different data types (e.g. tabular data, images, audios, texts, etc.). For all this reason, deep learning is attracting more and more researchers and analysts in all fields.

Although being well-known for its power in data processing and statistical modeling, SAS is still relatively new as a deep learning toolbox. Advance types of deep learning models like Convolutional Neural Networks [1] and Recurrent Neural Networks [2] are only available in the new SAS Viya environment through the Cloud Analytic Services (CAS). On the other hand, Python has become "the language for deep learning" due to its flexibilities and great supports from deep learning researchers.

In this paper, we discuss a case study of combining SAS and Python in utilizing deep learning for analytical tasks. More specifically, we focus on using deep architectures for sequential data for the task of predicting next-day stock price direction. The models we use include vanilla Recurrent Neural Network (RNN) [2], Long Short-Term Memory(LSTM) [3], Gated Recurrent Unit (GRU) [4], and a novel model we proposed named Recurrent Embedding Kernel (REK) [5]. Based on the power of each framework, we preprocess data in SAS and build our deep models in Python. The whole process is unified in one framework using the SASPy [6] module, which allows access to SAS codes in Python environment. As we use Theano [7], a low-level deep learning library in Python which might be over-complicated for the general users, we will focus more on connecting a Python session to SAS and executing SAS codes in Python environment through SASPy. In the next session, we first review the three mentioned deep learning models for sequential data, namely RNN, LSTM, and GRU.

DEEP LEARNING FOR SEQUENTIAL DATA

Deep learning algorithms use a vast number of parameters stacked by layers to model the data. The simplest form of a deep network is a deep feed-forward network (or deep neural network - DNN) [8]. Let H_i, W_i, and b_i denote the output, the weight matrix, and the bias vector of hidden layer i^{th} respectively, then

$$H_{i+1} = \sigma(W_i \cdot H_i + b_i) \tag{1}$$

with $\sigma(\cdot)$ being an activation function, often in the form of sigmoid, hyperbolic tangent, or rectified linear function (ReLU). The output layer of a DNN uses a task-driven output function, e.g., SoftMax for classification tasks, or a linear function for regression tasks. Figure 1 shows a common illustration of a DNN. The nodes refer to the outputs of the layers, the last nodes in each row represent the bias of the layers, and the connections between nodes represent to the weights of the layers.

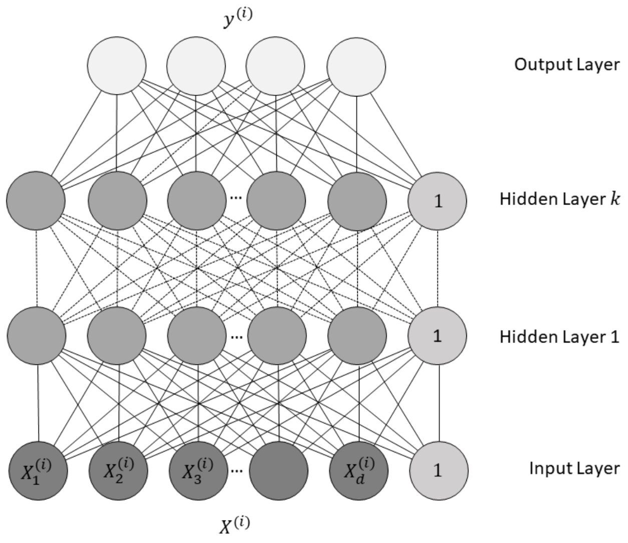


Figure 1. A Common Illustration of a Deep Neural Network

DNNs are usually trained to minimize a predefined loss function L using gradient descent. In details, with a loss function L defined, the network is iteratively updated by

$$\begin{aligned} W_i &\leftarrow W_i - \alpha \frac{\partial L}{\partial W_i} \\ b_i &\leftarrow b_i - \alpha \frac{\partial L}{\partial b_i} \end{aligned} \tag{2}$$

where α is the learning rate. Commonly, L varies by the task given to the network. For example, a binary classification DNN uses the binary cross-entropy loss function, a multi-label classification DNN uses the negative log likelihood loss function, and a regression DNN uses the mean squared error loss function.

Recurrent Neural Networks (RNN) are specifically designed to handle temporal information in sequential data. Commonly used RNN types include RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). In vanilla RNN's, the memory state of the current time point is computed from both the current input and its previous memory state. More formally, given a sequence $X = \{X_0, X_1, \dots, X_T\}$, the hidden state U_t of X_t (i.e. the state of X at time t) outputted by the network can be expressed as

$$U_t = \sigma(W \cdot X_t + R \cdot U_{t-1} + b) \tag{3}$$

where W and R are weight matrices of the network; b is the bias vector of the network; and $\sigma(\cdot)$ is a selected activation function. The computational flow of RNN is shown in Figure 2.

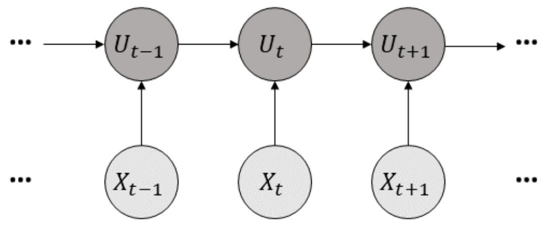


Figure 2. The Computational Flow of RNN

Since its memory state is updated with the current input at every time point, vanilla RNN is typically unable to keep long-term memory. LSTM is an improved version of RNN with the design goal of learning to capture both long-term and short-term memories. A LSTM block, shown in Figure 3, uses gates to control how much its long-term memory would be updated at each time point. The outputted short-term memory is then computed from the current input, the current long-term memory, and the previous short-term memory. More formally, an LSTM block can be described by the following formula:

$$\begin{aligned}
Z_t &= g(W_Z \cdot X_t + R_Z \cdot U_{t-1} + b_z) \\
i_t &= \sigma(W_i \cdot X_t + R_i \cdot U_{t-1} + p_i \times C_{t-1} + b_i) \\
f_t &= \sigma(W_f \cdot X_t + R_f \cdot U_{t-1} + p_f \times C_{t-1} + b_f) \\
C_t &= i_t \times Z_t + f_t \times C_{t-1} \\
o_t &= \sigma(W_o \cdot X_t + R_o \cdot U_{t-1} + p_o \times C_{t-1} + b_o) \\
U_t &= o_t \times h(C_t)
\end{aligned} \tag{4}$$

where W_* and R_* are weight matrices; b_* are bias vectors; p_* are peepholes; X_t, U_t, and C_t are the LSTM's input, output, and cell state (i.e. long-term memory) at time point$t$; Z_t is the proposed update to the cell state; i_t, f_t, and o_t are the output of the input gate, forget gate, and output gate, respectively; $g(\cdot)$ is the input activation, $\sigma(\cdot)$ is the sigmoid function, and $h(\cdot)$ is the output activation. The overall architecture and computational flow of LSTM is shown in Figure 3. As can be seen, the current long-term memory is computed from the previous input, short-term memory, and long-term memory, and the current short-term memory is computed using the current long-term memory. Unlike the vanilla RNN, all the memory updates are controlled using gates (represented by the dashed arrows in Figure 3).

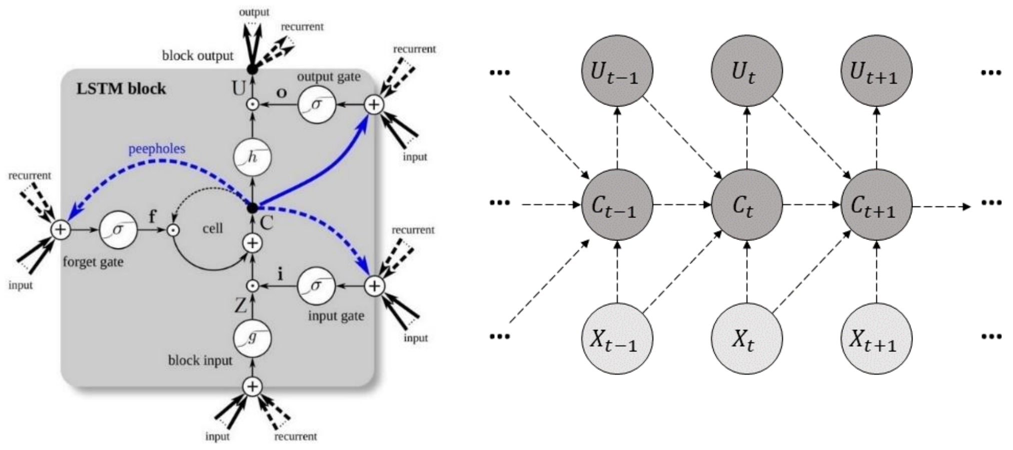


Figure 3. The Architecture[1] and Computational Flow of LSTM

[1] Figure adapted from [3]

Compared with vanilla RNN, LSTM introduces a mechanism to learn to capture task-relevant long-term memory. At each time point, the captured long-term memory is expressed as a vector. However, the architecture of an LSTM block is relatively complex, which may cause training of a LSTM-based model difficult and time consuming. GRU can be viewed as an alternative to LSTM that can learn to capture task-relevant long-term memories with a simplified architecture. A GRU block contains only two gates, as shown in Figure 4. It can be mathematically described using the following formula:

$$\begin{aligned} U_t &= (1 - z_t) \times U_{t-1} + z_t \times \widetilde{U_t} \\ \widetilde{U_t} &= g(W_U \cdot X_t + R_U \cdot (r_t \times U_{t-1}) + b_U) \\ z_t &= \sigma(W_z \cdot X_t + R_z \cdot U_{t-1} + b_z) \\ r_t &= \sigma(W_r \cdot X_t + R_r \cdot U_{t-1} + b_r) \end{aligned} \tag{5}$$

where all notations are similar to LSTM, except for z_t and r_t, which are the outputs of the update gate and reset gate, respectively. The computational flow of GRU is similar to that of vanilla RNN in Figure 2, except that the memory update is also controlled with gates like LSTM.

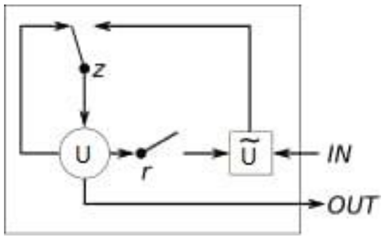


Figure 4. The Architecture of a GRU Block[2]

The final deep architecture we utilize in this paper is the Recurrent Embedding Kernel (REK) which is proposed in [5]. Unlike the other three recurrent architectures that make decisions directly from the short-term memory states, REK has another component that takes inputs as the short-term memory states at each pair of time points and compute their pairwise similarity. Decisions are then made based on the computed similarities using a K Nearest Neighbors (KNN) strategy. To do this, REK consists of a recurrent embedding network (REN) and a similarity network (SN). The REN component takes input state at each time point to compute their memory states that are represented through embedding vectors. In this case, any of the mentioned recurrent architectures can be used in place of REN. Each pair of memory states is then aggregated to form a single vector that is fed into the SN component to compute the pair's similarity. More specifically, SN is a deep network that output one single scalar representing the pairwise similarity of the time points. We use the sigmoid function as the output function of SN so that the similarity of a pair of instances can be interpreted as the probability of them having the same labels.

We show the computational flow and the decision-making process of REK in Figure 5. With the KNN strategy, REK makes decision at each time point based on the whole historical sequence before it. Mathematically, let $t < v$ be two time points in the sequence, $\hat{E}(\cdot)$ be the vectorizing function represented by the REN component, and $\hat{S}(\cdot)$ be the SN component, then we have

$$\begin{aligned} U_t &= \hat{E}(X_t, U_{t-1}) \\ U_v &= \hat{E}(X_v, U_{v-1}) \\ U_{(t,v)} &= U_t \odot U_v \\ s(X_t, X_v) &= \hat{S}\big(U_{(t,v)}\big) \end{aligned} \tag{6}$$

[2] Figure adapted from [4]

where

$$U_{(t,v)} = \{U_{t1} * U_{v1}, U_{t2} * U_{v2}, \dots U_{td} * U_{vd}, |U_{t1} - U_{v1}|, |U_{t2} - U_{v2}|, \dots, |U_{td} - U_{vd}|\} \quad (7)$$

and $s(X_t, X_v)$ represents the pairwise similarity between X_t and X_v. When making prediction for a time point with an unknown label, we first compute its similarity with the rest of the time points (with known labels). Then, we base on the computed similarities to find most similar time points of the unknown one, and assign its label accordingly.

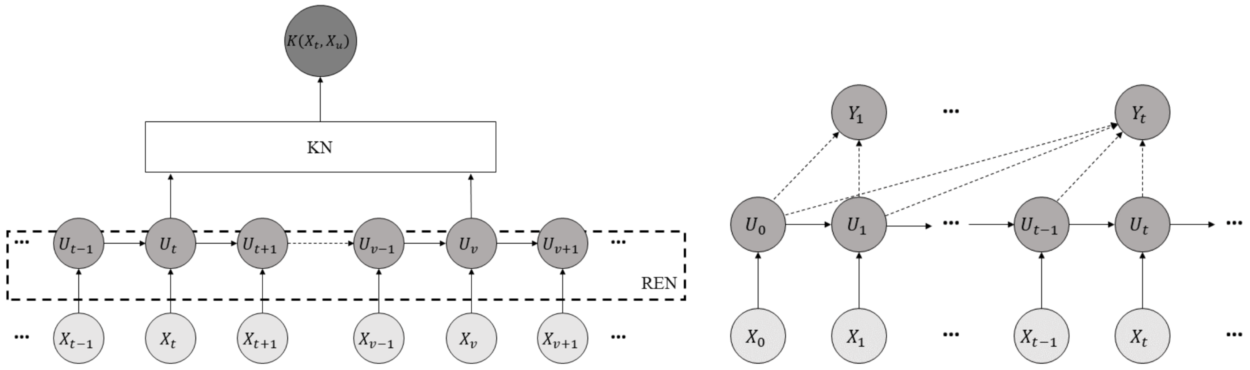


Figure 5. The Computational Flow and Decision-Making Process of Recurrent Embedding Kernel

To ensure both networks of REK are trained towards the goal of optimizing the decision making for KNN, we sample training data in the neighborhood of each instance. More specifically, each iteration in the training process of REK on a sequence $X = \{X_0, X_1, \dots X_t, \dots\}$ can be described as follows.

1. Input the whole sequence into REN to obtain embedding vectors from each time point
2. Input all pair of embedding vectors into the KN to obtain their pairwise similarities
3. Use the computed similarities to determine the neighborhood of k nearest time points with the same label for each time point in the training sequence
4. Compute the loss based on pairs of time points and all time points in their determined neighborhood
5. Use Gradient Descent to update the weight matrices and bias vectors throughout the REK network to minimize the aggregated loss

CASE STUDY: PREDICTING DAILY STOCK PRICE DIRECTION

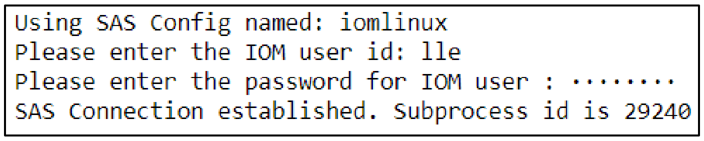
In this section, we utilize all discussed deep models into the specific task of predicting daily stock price direction. More specifically, given the price sequence of a stock ticker up to the current day, we predict if the adjusted close price would be up or down for the next trading day. Utilizing recurrent architectures, the prediction not only depends on the price of the current day but also on certain critical information from historical price movements.

As discussed, we implement all experiments Python 2.7.15rc1 with the use of the SASPy library to execute SAS codes in Python environment. Assuming the SASPy package is already installed and configured to connect to the user's SAS server, a SAS session is started by importing the SASPy library and call the *SASSession* function:

```
import saspy
#additional packages to process data
import pandas as pd
import numpy as np
```

```
from IPython.display import HTML

#initialize SAS session
sas = saspy.SASsession()
```

Here, we also import the Pandas [9] and Numpy [10] packages to use later. Additionally, we import the HTLM module to show results from SAS code in HTML format. If SASPy is configured correctly, the users will be required to log in. A successfully initialized session shows output as showed in Figure 6. The created *sas* object then represents the SAS session in Python.



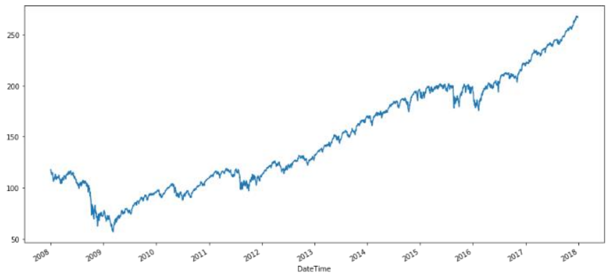
```
Using SAS Config named: iomlinux
Please enter the IOM user id: lle
Please enter the password for IOM user : ········
SAS Connection established. Subprocess id is 29240
```

Figure 6. Logging into a SAS Session with Saspy in Jupyter Notebook

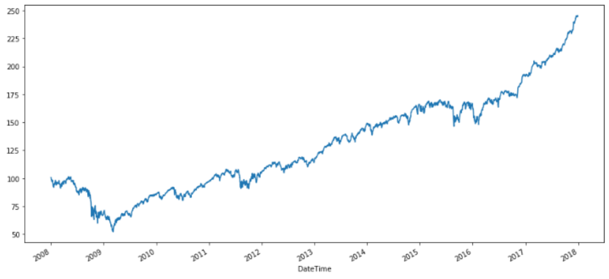
With a SAS session established, we begin our analysis with the stock data. All stock data are obtained from Yahoo! Finance from the beginning of 2008 to the end of 2017. We test our model using the daily, weekly, and monthly data from six ETFs:

1. SPDR S\P 500 ETF Trust (SPY)
2. SPDR Dow Jones Industrial Average ETF (DIA)
3. iShares US real Estate ETF (IYR)
4. SPDR Gold Shares (GLD)
5. Vanguard Energy ETF (VDE)
6. VanEck Vectors Gold Miners ETF (GDX)

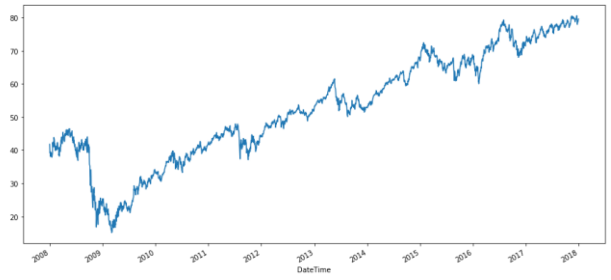
We select the six ETFs in order to cover different types of long-term trends. As shown in **Figure 7**, SPY, DIA, and IYR have an overall upward trend; GLD presents an uptrend followed by a downtrend; VDE develops a long-term consolidation; and GDX demonstrate a downward trend for the most part of the series.

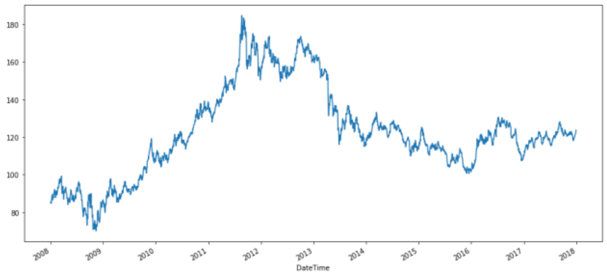


a) SPY



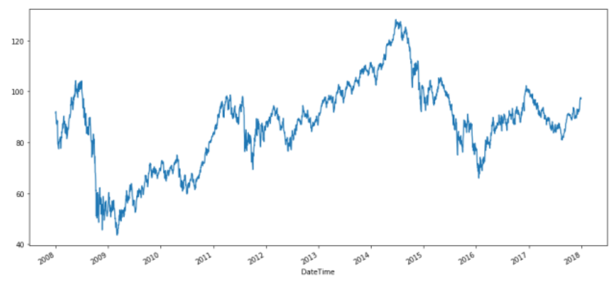
b) DIA

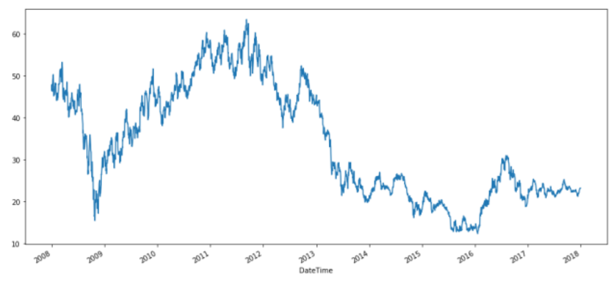




c) IYR

d) GLD





e) VDE

f) GDX

Figure 7. The Trends of Experimented Stock Series

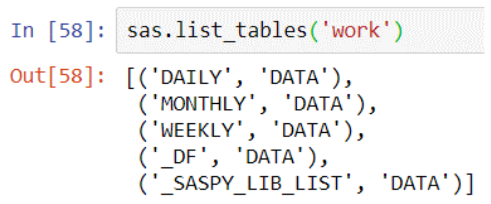
Data downloaded from Yahoo! Finance is in CSV format, and the daily, weekly, and monthly data are in separate files. Unless the data is already available on the SAS server, the users need to load the downloaded data to the server from their local machine. The code snippet below loads the downloaded CSV into a Pandas data frame in Python, then use the $df2sd$ method of sas to send them to the server:

```
daily_pd = pd.read_csv("Daily.csv")
daily = sas.df2sd(daily_pd,table='daily')
weekly_pd = pd.read_csv("Weekly.csv")
weekly = sas.df2sd(weekly_pd,table='weekly')
monthly_pd = pd.read_csv("Monthly.csv")
monthly = sas.df2sd(monthly_pd,table='monthly')
```

After executing the snippet, the three sets are available in the SAS server as SAS datasets; their names can be set through the $table =$ argument in the $df2sd$ method. The users can check the available in a SAS library with the $list\_tables$ method. For example, the snippet below lists all datasets in the WORK library:

```
sas.list_tables('work')
```

which shows results as in Figure 8.



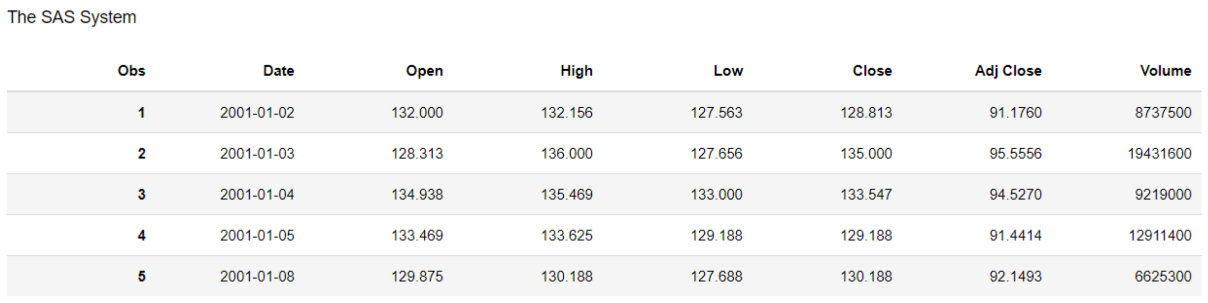
```
In [58]: sas.list_tables('work')

Out[58]: [('DAILY', 'DATA'),
          ('MONTHLY', 'DATA'),
          ('WEEKLY', 'DATA'),
          ('_DF', 'DATA'),
          ('_SASPY_LIB_LIST', 'DATA')]
```

Figure 8. Output from the list_table Method

With the data loaded, the users can use different methods or functions available through the sas object to analyze the data with the connected SAS system. Alternatively, the user can user the $submit$ method of the sas object to directly submit SAS code in Python. For example, the code snippet below can be run from Python to submit the PRINT procedure in SAS to display the first five observations of the daily dataset:

```
c = sas.submit("""
    proc print data=daily (obs=5);
    run;
""")
HTML(c['LST'])
```

The result of the snippet above is identical to results from a PRINT procedure executed in SAS environment, as showed in Figure 9.



The SAS System

Obs	Date	Open	High	Low	Close	Adj Close	Volume
1	2001-01-02	132.000	132.156	127.563	128.813	91.1760	8737500
2	2001-01-03	128.313	136.000	127.656	135.000	95.5556	19431600
3	2001-01-04	134.938	135.469	133.000	133.547	94.5270	9219000
4	2001-01-05	133.469	133.625	129.188	129.188	91.4414	12911400
5	2001-01-08	129.875	130.188	127.688	130.188	92.1493	6625300

Figure 9. Output of the PRINT Procedure Called in Python through Saspy

Our task is to predict if the adjusted close price of the next day is up or down. For this purpose, we label each day in the training sequence as "Up" if the adjusted close of its next day is higher than its current adjusted close price; otherwise, we label it as "Down". We use 18 input features to describe each day, which include open, close, highest, lowest, adjusted close, and volume, of the current day, current week, and current month. The code snippet below modifies the imported data from Pandas, then sort and merge them into a dataset named "*all*" that contains all daily, weekly, and monthly information:

```
c = sas.submit("""
/**Import data and rename daily, weekly, and monthly data**/
data daily;
    set daily;
    date_num = input(date, yymmdd10.);
    format date_num yymmdd10.;
    rename open = open_d close = close_d high = high_d low = low_d
                  'adj close'n = adjclose_d volume = volume_d;
data weekly;
    set weekly;
    date_num = input(date, yymmdd10.);
    format date_num yymmdd10.;
    rename open = open_w close = close_w high = high_w low = low_w
                 'adj close'n = adjclose_w volume = volume_w;
data monthly;
    set monthly;
    date_num = input(date, yymmdd10.);
    format date_num yymmdd10.;
    rename open = open_m close = close_m high = high_m low = low_m
                 'adj close'n = adjclose_m volume = volume_m;
/**Merge**/
data daily;
    set daily;
    week = WEEK(date_num); month = MONTH(date_num); year = YEAR(date_num);
proc sort data=daily; by year week;
data weekly;
    set weekly;
    week = WEEK(date_num); month = MONTH(date_num); year = YEAR(date_num);
proc sort data=weekly; by year week;
data monthly;
    set monthly;
    week = WEEK(date_num); month = MONTH(date_num); year = YEAR(date_num);
proc sort data=monthly; by year month ;
data all;
```

```
    merge daily weekly; by year week;
proc sort data=all; by year month;
data all;
    merge all monthly; by year month; """)
```

Given that the stock price can change dramatically over a long-term period, in order to avoid the sharp differences in the input values of any two given days, we use the differences of current day's values and previous day's values as the input values for each feature. This is done with the code snippet below:

```
c = sas.submit("""
/**Get Difference**/
data diff;
    set all;
    open_d = dif(open_d); close_d = dif(close_d);
    high_d = dif(high_d); low_d = dif(low_d);
    adjclose_d = dif(adjclose_d); volume_d = dif(volume_d);
    open_w = dif(open_w); close_w = dif(close_w);
    high_w = dif(high_w); low_w = dif(low_w);
    adjclose_w = dif(adjclose_w); volume_w = dif(volume_w);
    open_m = dif(open_m); close_m = dif(close_m);
    high_m = dif(high_m); low_m = dif(low_m);
    adjclose_m = dif(adjclose_m); volume_m = dif(volume_m);
    ;
""")
```

Next, we generate the labels for the data and merge them with the differenced features to form a master dataset, then standardizing all features:

```
c = sas.submit("""
/**Get Label**/
data Y;
    set diff;
    date_num = lag(date_num);
    if adjclose_d > 0
    then y = 1;
    else y = 0;
    keep date_num y;
proc sort data=diff; by date_num;
proc sort data=y; by date_num;
data master;
    merge diff y; by date_num;
proc standard data=master mean=0 std=1 out=master_std;
    var open_d close_d high_d low_d adjclose_d volume_d
        open_w close_w high_w low_w adjclose_w volume_w
        open_m close_m high_m low_m adjclose_m volume_m;
""")
```

Finally, the processed dataset in SAS can be loaded back to the local Python system with the *sasdata2dataframe* method of the *sas* object:

```
data = sas.sasdata2dataframe('master_std')
```

In Python, we split data from 2008 to 2016 to for training, and data in 2017 for testing; the training set is further split into training (2008 to 2015) and validation (2016) for determining early stop of the training:

```
in_list = ['open_d', 'high_d', 'low_d', 'close_d', 'adjclose_d',
           'volume_d', 'open_w', 'high_w', 'low_w', 'close_w',
           'adjclose_w', 'volume_w', 'open_m', 'high_m',
           'low_m', 'close_m', 'adjclose_m', 'volume_m']

X = data[['date_num'] + in_list]
y = data[['date_num','y']]

trainX = X[X['date_num']<'01-01-2015'][in_list].values
validX = X[(X['date_num']>='01-01-2015') & (X['date_num']<='01-01-
2016')][in_list].values
testX = X[X['date_num']>='01-01-2017'][in_list].values

trainY = y[y['date_num']<'01-01-2015']['y'].values
validY = y[(y['date_num']>='01-01-2015') & (y['date_num']<='01-01-
2016')]['y'].values
testY = y[y['date_num']>='01-01-2017']['y'].values
```

We train REK's using the discussed strategy. The REK that is used in all experiments includes a two-layer REN (with 36 hidden neurons each layer), and a three-layer KN (one hidden layer of 36 neurons and one output neuron). LSTM and GRU models have 36 hidden neurons. Vanilla RNN models have two hidden layers of 36 neurons. All referenced models use Softmax for decision making. All models are trained with decaying learning rates starting from 0.1. We then decrease the learning rates by a factor of 10 when training cost begins to heavily fluctuate among adjacent epochs. Model training is controlled using early stopping with validation accuracy. The last model with highest validation accuracy is then fitted on the testing data to obtain the testing accuracy. All models use ReLU activation, except for LSTM that uses tanh, because ReLU LSTMs explode rapidly in our experiments.

Table 1 shows the testing performance of all models on each ETF. We also include the percentage of actual Up days in the testing data in the second column of Table 1.

Table 1. Classification Accuracy of Tested Models

Data	% Up	REK	RNN	GRU	LSTM
SPY	57.32	71.43	67.55	69.11	69.11
DIA	61.45	71.89	67.55	70.27	68.07
IYR	53.01	70.06	67.47	65.86	69.08
GLD	56.63	70.28	67.46	66.36	70.04
VDE	47.79	71.89	64.23	69.88	70.28
GDX	50.6	67.47	61.45	55.02	60.24

We are not including the Python code to build, train, and test, the deep models in this paper since they are implemented in Theano which is a low-level deep learning package. More specifically, building a deep network in Theano requires processes such as initializing all variables and parameters, building their computational flow, and manually execute Gradient Descent, to be done manually. Therefore, the Theano code for each of these models is relatively complicated and might not be of interests to the general users. With the purpose of using common deep learning models at the application level, we recommend the uses of high-level deep learning tools like TensorFlow [11], Keras [12], or PyTorch [13]. In such packages, the users can quickly generate a network with pre-defined functions where hyper-parameters like network types, number of layers, layer types, etc., are directly input.

CONCLUSION

In this paper, we showcase an analysis that is done with a combination of SAS and Python with the aids of the saspy package. In more details, saspy connects a Python session to a SAS server, and allows users to submit SAS code and use the server backend in their Python session. Based on each tool's advantages, we use SAS to preprocess the data, and Python to build and fit deep learning models, on the task of predicting daily stock price direction.

This paper can also be used as a general case study of connecting between a SAS session and a Python session to utilize their different toolboxes. For example, SAS statistical modeling, or Python machine learning packages (e.g. Sci-kit Learn [14]).

REFERENCES

[1] LeCun, Yann, Yoshua Bengio, et al. (1995). "Convolutional networks for images, speech, and time series". In: The handbook of brain theory and neural networks 3361.10, p. 1995.

[2] Funahashi, Ken-ichi and Yuichi Nakamura (1993). "Approximation of dynamical systems by continuous time recurrent neural networks". In: Neural networks 6.6, pp. 801–806.

[3] Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: Neural computation 9.8, pp. 1735–1780.

[4] Chung, Junyoung et al. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: arXiv preprint arXiv:1412.3555.

[5] Le, Linh and Ying Xie (2018). "Recurrent Embedding Kernel for Predicting Stock Daily Direction". In: 2018 IEEE/ACM 5rd International Conference on Big Data Computing Applications and Technologies (BDCAT).

[6] Saspy 2.4.3 Documentation, sassoftware.github.io/saspy/.

[7] Bergstra, James et al. (2010). "Theano:ACPU and GPU math compiler in Python". In: Proc. 9th Python in Science Conf, pp. 1–7.

[8] Schmidhuber, Jürgen (2015). "Deep learning in neural networks: An overview". In: Neural networks 61, pp. 85–117.

[9] McKinney, W. (2011). pandas: a foundational Python library for data analysis and statistics. Python for High Performance and Scientific Computing, 14.

[10] Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2), 22.

[11] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16) (pp. 265-283).

[12] Gulli, A., & Pal, S. (2017). Deep Learning with Keras. Packt Publishing Ltd.

[13] Ketkar, N. (2017). Introduction to pytorch. In Deep learning with python (pp. 195-208). Apress, Berkeley, CA.

[14] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.

CONTACT INFORMATION

Linh Le
Institute of Analytics and Data Science
Kennesaw State University
lle12@students.kennesaw.edu