Midas: a Python Framework for Automated Generating and Training of Neural Network Models

Submission Type: Demo/Prototype

Nikolaj Krebs Pedersen
nped@itu.dk
IT University of Copenhagen
Denmark

Anders Widtfeldt Meged
andersmeged@gmail.com
IT University of Copenhagen
Denmark

Rune André Johansen
rujohansen@kpmg.com
IT University of Copenhagen
Denmark

Abstract

As social interactions increasingly take place in digital environments, a vast and increasing volume of digital traces in the form of unstructured textual data is produced. Currently, the machine learning techniques to analyze such data require deep knowledge of machine learning, usually confined to highly specialized data scientists and statisticians. This high competence threshold excludes a large group of businesses and data analysts from leveraging machine learning and creating value from unstructured digital trace data.

This paper shows how the Midas framework can help data analysts analyze unstructured text data using neural networks. It outlines the framework's main features and provides example guides on how to implement the Midas framework on specific datasets. We hope this framework will make it easier and more accessible for data analysts working with unstructured text data to leverage neural networks.

Motivation

This prototype paper presents the Midas Framework, a Python framework that allows users to automatically generate and train a TensorFlow Neural Network [NN] model. The Midas framework allows for a wider accessibility and broader implementation of NNs within businesses by automating the highly technical model generating process. This is achieved through simplifying the process of generating NN models and implementing aspects of no-code to support data analysts in leveraging NN. The framework is developed to leverage tabular and unstructured textual data to generate patterns in the vast and growing volume of unstructured data being produced in a plethora of different contexts. This first version of the framework is developed with the intent to support the less technical role of data analysts. The definition of a data analyst is distinguished from a data scientist by focusing on speed and targeting business intelligence. Data science requires competence in one or more of three domains: statistics, Machine Learning [ML], and analysis (Kozyrkov, 2018) Statisticians are experts in arriving at conclusions beyond the data in a safe and structured manner and focus intensely on whether the method is appropriate for the problem. ML or AI engineers focus on the solution's accuracy regardless of time to achieve engineering excellence. Data analysts focus on quickly finding interesting trends within given data (Kozyrkov, 2018). The Midas Framework is developed to support the latter aspect of data science. By adding to existing work on democratization of ML tools, we hope to enable more data analysts to leverage NN.

Overview of the Midas Framework

The Midas Framework consists of two classes: *"Midas_Generate"* and *"Midas"*. The Midas class will be the primary class with which users will interact. It consists of 8 functions that users can use to generate an NN model, use the model for prediction, and save and load a model.

Midas.Generate() - This function generates, trains and delivers a neural network for a given dataframe. Here this class will use the Midas_Generate.main() function. The .Generate() function takes several attributes that are used to generate a NN model (Table 1). When calling the function, these attributes are defined by the user. The only mandatory attributes are: "dataset", "vect", "y_var", "x_var", and "NLP_var". The other, if not specified, will use standardized configurations (Figure 1).

```
Midas.Generate(dataset=df, vect='vect_file.vec', y_var='dependent_variable', x_var='independent_variable',
               NLP_var='independent_variable_textual', t_var='n', LSTM_var = '128 64 256',
               DL_var = '20 30 50', o_var = 'adam', NN_mode = "regression", bs_var = 128,
               e_var = 10, ver_var = 1, val_var = 0.2)
```

Figure 1. Midas.Generate() function

The *Midas_Generate.main()* function goes through and applies given attributes within a flow of local functions of the *Midas_Generate* class. Firstly, it creates a subset of the given dataframe, using the defined columns from the attributes of; *"y_var", "x_var"*, and *"NLP_var"*. The model will only test and train based on the user's defined columns. Then the subset is split into train and test data using the scikit-learn method "*train_test_split()"*. The data is then made numeric through a function that utilizes *keras.preprocessing.LabelEncoder()*. This function takes all string values, except the subset's data defined by *NLP_var*, and changes them into numeric values.

The Midas Framework utilizes tokenization to convert unstructured textual data into input data. This is done by preprocessing the textual data and then

utilizing *TensorFlow.keras.preprocessing.text. Tokenizer ()* to create the data into smaller units of sentences and transform them into tokens. These are then able to be used as input for the NN model. Furthermore, which is generated by an internal function of the *Midas_Generate* class that creates an embedding matrix based on the *.vec* file defined by the *vect* argument.

Lastly, the *.main()* function uses the data to generate a NN model. It consists of two TensorFlow models. First, it creates a Sequential model with architecture for prediction based on textual data. Its architecture consists of an Input-layer, Embedding-layer, a number of LSTM layers determined by *LSTM_var*, and a flatten layer. This model is then concatenated with another Sequential model that contains an input layer for tabular data and a number of dense layers, defined by *DL_var,* with the "relu" activation function. Together these two models form the generated Midas NN model. BatchNormalization layers are then added to normalize data, and Dropout-layers are added to support the model in not overfitting. The attribute NN_mode defines the output layers of the NN model. This attribute takes three different inputs *"regression", "binary",* and *"categorical "*. Based on this input, the output layer is created along with finalizing the data argumentation. The regression makes a single output neuron without any activation function and sets the loss function to be MeanSquaredError as the output. The binary mode creates another single output neuron containing a "sigmoid" function as an activation function, setting the loss function to binary_crossentrophy. The categorical model creates an output layer equal to the number of categories, and one hot encodes the target variable in order to make it into an equal representation of the number of output neurons and sets the loss function to categorical_crossentropy.

The model is lastly trained using the train datasets with an optimizer, batch size, epoch number, verbosity level, and a validation split defined by attributes given by the user. To train, the metrics

used are also determined by *NN_mode*. Both categorical modes use the metrics of "accuracy", and the regression mode uses "mean squared error".

After completing the *Midas_Generate.main(),* the tokenizer, the maxlen, the trained NN model, and the NN_mode used are returned. Here they are saved to local variables of the instanced *Midas* class.

Midas.SaveModel() - saves a Midas model to the current directory saving it for future use. It takes one attribute, "*name"*, which will determine the name of the saved model. This function is also dependent on a model that has been generated, as it is saved as a local variable (Figure 2)

Midas.LoadModel() - This saved model can then be accessed through the *.LoadModel()* function. Here the user needs to specify a name that the function then uses to locate and load a model from the current directory. Within the *Midas* class, the model is then saved as the local variable *"model"* (Figure 2).

```
Midas.SaveModel('Model_Name')

Midas.LoadModel('Model_Name.h5')
```

Figure 2. Midas.SaveModel() & Midas.LoadModel() function

Midas.Predict() **-** creates predictions based on the current model. The function takes three attributes *"dataset", "X_pred"*, and *"NLP_pred"* (Table 1). The data defined by the attributes are then used by *keras.tensorflow.predict()* method to predict upon, returning predictions that the user then can access.

```
Midas.Predict(dataset=df, X_pred='predictive_independent_variables', NLP_pred='predictive_independent_variable_textual',
              NN_mode='regression')
```

Figure 3. Midas.Predict() function

Table 1. Overview of argument/parameters of Midas functions

Parameter name	Parameter description
dataset	Any valid Pandas dataframe object.
vect	Any valid file path to a .vec file containing a vectorization map of words.
y_var	A string variable specifying any numerical or categorical column from the Pandas object delivered from the dataset parameter and sets it to the model's target variable.
X_var	A string variable specifying one or multiple categorical or numerical columns from the Pandas object delivered by the DataFrame parameter and set the predictor variables. A comma separates the string.
NLP_var	A string variable specifies the specific column containing unstructured textual data, submitting it for textual preprocessing and shaping the model input according to the average length of text.
sc_var	A string variable defining which of the specific variables from x_var, should be scaled during preprocessing. A comma-separated string specifies the variables.
t_var	A string variable which decides whether the model should be trainable. This parameter functions as a boolean and can be set to yes = "y" or no = "n".
LSTM_var	A string variable that specifies the depth and density of the model's LSTM (Long Short Term Memory) architecture. A comma-separated string specifies the variables.
DL_var	A string variable that specifies the depth and density of the perceptrons in the model. A comma-separated string specifies the variables.
o_var	A string variable that specifies any optimizer available in Tensorflow 2.10.0
s_var	A String variable defining which scaler should be used on the variables specified for scaling can be set to "MinMaxScaler" or "Standardscaler" provided by Scikit-learn.
NN_mode	A string variable specifying the type of prediction the target variable should be set for. Different modes include "binary", "categorical" and "regression".
bs_var	An integer specifying the batch size of observations for training.

e_var	An integer specifying the number of epochs the model should iterate training over the training data.
ver_var	A categorical integer which determines the TensorFlow verbosity level allowing the user to grant control over the training log. If not specified, the default will be set to 1, allowing the user to follow the training of the models through the epochs.
val_var	An integer specifying the training split during the model training for internal validation.

Requirements

For the framework to function, users must have preliminary knowledge of the Pandas API and how to operate it. Knowledge about how to preprocess data is also needed as the Midas Framework does not support its users in conducting exploratory data analysis (EDA) or data manipulation. That said, it includes a label encoder that ensures the input for the NN model is numeric. A *.vec* file with pre-trained word embeddings is also required for the Midas Framework to create word embeddings used to interpret the unstructured tabular data through tokenization. The file needed will be context-specific to the problem, and the language of the unstructured textual data.

Installation

The Midas Framework can be accessed through the following link:

https://github.com/MidasFramework/Midas_Framework

Located within the GitHub page are three files: *"INSTALL.txt"," Midas_Framework_v1.py",* and *"README.txt"*. These files describe the Midas Framework's installation process, including the needed packages.

Guide and Examples of the Midas Framework

The following section will present a guide describing how users interact with the Midas Framework. The guide will focus on the steps needed to take after the completion of the section

Installation where the required packages are imported and the necessary data exploration and manipulation have been concluded.

After Midas and the additional libraries have been imported, the user creates an instance of the Midas class as an object to access its functions (Figure 4).

```
1  m = Midas()
```

Figure 4 – Example of the creation of an instance of the Midas class

The first step of providing the user with a generated model is running the function *.Generate()*. The function takes five mandatory attributes: a Pandas dataframe, a .vec file, and definitions of the dependent- and independent variables. The input for the dependent variable is divided into two attributes; one for structured tabular data (*x_var*) and the other for unstructured textual data (*NLP_var*). The user will also have to determine the NN mode dependent on the problem they wish to predict (Figure 5). The user is also able to manually configure hyperparameters of the NN model through the attributes described in Table 1. If these are not defined by the user, a standard configuration is used.

```
1  m.Generate(dataset=df,vect='cc.en.300.vec', y_var='Statistics',
2             x_var='ABSTRACT, Computer Science, Physics, Mathematics, Quantitative Biology, Quantitative Finance',
3             NLP_var='ABSTRACT',NN_mode='binary')
```

Figure 5 – Example of the Midas.Generate() function

After the .Generate() function is initialized the model generation will start, which will prompt the printed message "start" in the log. As part of the model generation, the provided data will go through the necessary data augmentation to ensure that the data is prepared and can be used to train the generated model. The data augmentation includes splitting the data into train and test, labelling of string data points - excluding the data used for unstructured data, and tokenizing

unstructured data. The function then creates the model architecture, which the framework presents in the log through the *model.summary()* function from the TensorFlow framework. Following the printed model summary, the training of the model will initialize. The training process is visualized through verbosity level. The speed at which the training goes through the determined epochs is shown, including information regarding loss and accuracy for each epoch. The .*Generate()* function ends by returning a NN model saved as a variable in the created Midas Object. To utilize the model for predictions, users can load data into a separate Pandas dataframe and specify equivalent data parameters into the Midas class. *Predict()* function will return an array of predictions from the model to the user (Figure 6).

```
1 m.Predict(dataset=df, X_pred='Computer Science, Physics, Mathematics, Quantitative Biology, Quantitative Finance',
2           NLP_pred='ABSTRACT', NN_mode='Binary')
656/656 [==============================] - 200s 302ms/step

array([[0.18440267],
       [0.11621204],
       [0.0558403 ],
       ...,
       [0.09753618],
       [0.6240082 ],
       [0.85602134]], dtype=float32)
```

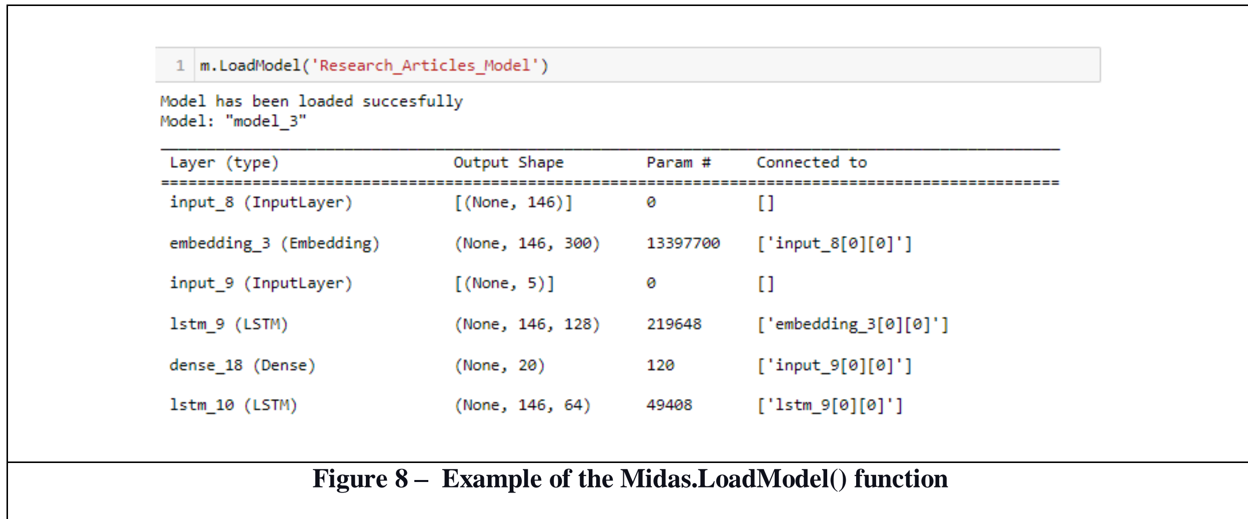
Figure 6 – Example of the Midas.Predict() function

Through the Midas function .*SaveModel()* the users can save the generated model and its weights to their directory, enabling them to access the model later (Figure 7).

```
1 m.SaveModel('Research_Articles_Model')
Model have been saved Succesfully
```

Figure 7 – Example of the Midas.SaveModel() function

To access the saved model, users would utilize the Midas function .*LoadModel()*. To make sure the correct model is loaded a visualization of the model architecture is shown (Figure 8).



```
1 m.LoadModel('Research_Articles_Model')

Model has been loaded succesfully
Model: "model_3"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_8 (InputLayer)            [(None, 146)]        0           []

embedding_3 (Embedding)         (None, 146, 300)     13397700    ['input_8[0][0]']

input_9 (InputLayer)            [(None, 5)]          0           []

lstm_9 (LSTM)                   (None, 146, 128)     219648      ['embedding_3[0][0]']

dense_18 (Dense)                (None, 20)           120         ['input_9[0][0]']

lstm_10 (LSTM)                  (None, 146, 64)      49408       ['lstm_9[0][0]']
```

Figure 8 – Example of the Midas.LoadModel() function

Conclusion and Next Steps

This prototype paper demonstrates how the Midas Framework can help data analysts leverage Neural Networks to analyze unstructured textual data. It gives an overview of the main functions of the framework and provides an exemplified guide to implementing the Midas Framework on a specific dataset. It is our hope that the framework will make NN more accessible to data analysts working on unstructured textual data.

Future work on the framework aims to go even further in making NN accessible by providing a user interface and methods supporting the installation process, to further eliminate the need for specific coding competencies and enabling even larger user groups to leverage NN.

Acknowledgements

We are grateful for the support and guidance from Jonas Valbjørn Andersen and would like to express our most profound appreciation.

References

Kozyrkov, C. 2018." What Great Data Analysts Do – and Why Every Organization Needs Them", *Harvard Business Review*, pp. 147-156. https://hbr.org/2018/12/what-great-data-analysts-do-and-why-every-organization-needs-them