

SlotMarketSQL

Abstract

This project aims to help day traders make data-driven decisions on investment and portfolio management by enabling them to access forecasted daily closing stock prices and volatilities through a user-friendly chatbot interface. Long Short-Term Memory (LSTM) and Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models provide 1-day forecasts of the closing prices and volatilities of S&P 500 stocks. The chatbot is designed to convert text into SQL queries using transformer encoders that are trained for slot prediction. The chatbot thus generates a table that answers the user's question about forecasts or current-day information.

Zack Horton, Virginia Maguire, Tanner Street, and Yutong Zhang
15.775 - Hands-on Deep Learning (Group A26)
March 11, 2023

Problem Description	2
Methods	2
LSTM Model	2
Network Architecture Overview	2
Walk-Forward Validation	3
GARCH Model	3
Slot-Filling with Transformers	3
Approach	4
Experiments to Optimize Forecasting Architecture	4
GARCH Model Parameters	5
Evaluating LSTM Model Performance	5
Output Data for SlotMarketSQL Application	5
Building Slot-Filling Data	5
Training Transformers	6
SlotParser - From Tokens to SQL	6
Streamlit Application	6
Results	7
LSTM Architecture	7
Number of Epochs	7
Input Sequence Length	7
Number of LSTM Units and Dropout Rate	7
Number of LSTM Stacks	8
Final Architecture Summary	8
LSTM Model Evaluation	8
Normalized MSE and MAE	8
Mean Directional Error	9
Slot-filling Accuracy	9
Lessons Learned	9
LSTM Model	9
Data Used	9
Architecture Decisions	10
SlotMarketSQL Application	10
Manually Labeling Inputs to SQL	10
Filtering SQL Queries	10
Future Improvements	10
LSTM Model	10
GARCH Model	10
Transformer Model	11
Using LLMS	11
Works Cited	12
Appendix	13

Problem Description

The ability to leverage historical data and build intelligent models for future stock price and risk predictions is vital to day traders' decision-making process as well as portfolio management. This project aims to forecast daily closing stock prices and volatilities using deep learning models and create a chatbot that allows a day trader to easily access information by querying the interface.

Closing prices of S&P 500 stocks over two years (from 2022-01-22 to 2024-01-21) were used for model training and evaluation. The historical closing price data was procured from Yahoo Finance using the API.

Methods

LSTM Model

The LSTM model is a type of recurrent neural network (RNN) that can capture long-range relationships in sequence data, thereby alleviating the classic vanishing gradient issue. RNNs and LSTMs differ from feed-forward neural networks in that the output of a node can be fed back into the network, allowing the models to have a memory of previous information. LSTMs differ from RNNs due to their more complex architecture, which includes a cell state that controls information flow over long sequences much more effectively than RNNs. LSTMs are thus very effective at time-series forecasting since they can capture complex temporal dependencies over long sequences.

In this scenario, LSTMs are used to provide a next-day forecast of a stock's closing price, which is normalized for modeling purposes. The input is a sequence of the previous closing prices. The length of this sequence is an important parameter worth consideration. Longer sequences allow the model to learn more information and capture long-range relationships, but shorter sequences may be preferable when the long-range relationships are less important than the most recent information.

Network Architecture Overview

The input data must be structured as a 3D array. The first dimension refers to the number of sequences, the second dimension refers to the length of each sequence, and the third dimension refers to the number of features being forecasted. In this case, the number of features is one since this is a univariate time series.

The input data is fed to the LSTM layers after being properly shaped. An important parameter for LSTM models is the number of units. The number of units controls the dimensionality of the output vector and, by extension, the amount of information the model can learn. Additionally, LSTM layers can be stacked, so that the output of one

layer is the input to the next. Stacking layers allows the model to learn higher-level abstractions of the input. Dropout layers are typically used after LSTM layers to prevent overfitting. The dropout rate is thus an important parameter to consider in LSTM architectures. The output layer in this architecture has one node since the task is to predict a single value. The mean squared error (MSE) was used as the loss function.

Walk-Forward Validation

Since the models were trained to provide next-day forecasts, the validation sets only consist of one data point. Since one point is not adequately representative of a model's performance, walk-forward validation was used for evaluation. In this method, the models were trained up to a point and then evaluated on the subsequent point. The models were then retrained using the point previously used for validation and then evaluated on the subsequent point. This procedure repeats until the most recent point in the dataset is used for validation.

This method allows each model to be evaluated on more than just one point. Averaging the error values from each validation point provides a comprehensive overview of the model's performance.

GARCH Model

The GARCH model is a classical method used to capture heteroskedasticity, which is a non-constant variance, across a time series. These models predict variance based only on past variances (autoregressive component) and past squared residuals (generalized component). Each component corresponds to two parameters controlling how many previous values to use.

In this scenario, GARCH models were used to model stock volatility. These models were thus fit to the daily log returns, which were calculated via a simple transformation of the daily closing prices.

Slot-Filling with Transformers

As seen in class with the ATIS dataset example, a transformer model can assign elements of an inputted question to different tables within a database. This was accomplished through transformer encoders, which preserve the size of the input prompt and create an output vector of that same size. Therefore, a simple question can be taken as the prompt, and individual words can be associated with corresponding parts of an SQL query.

This particular project allows users to develop custom SQL queries to streamline analytics based on the stock market aggregate table. Once the tokenized output is

obtained, the "SlotParser" parses the transformer encoder's output and relays the requested information to the user.

Approach

Experiments to Optimize Forecasting Architecture

Before fitting the final models, a variety of experiments were conducted to determine an effective architecture. The following parameters were tested: epochs, input sequence length, number of LSTM units, dropout rate, and number of LSTM stacks. Three stocks were randomly selected and used for all experiments. Since all of the models share the same architecture, it is a reasonable assumption that three random stocks are adequately representative of the entire dataset.

The number of epochs was optimized to avoid underfitting and overfitting, with 1-20 epochs being tested. For each stock, walk-forward validation was used to evaluate the model's performance with varying numbers of epochs. The other components of the architecture were held constant. Since this was the first experiment conducted, the architecture parameters were selected based on a stock price prediction study (Gülmez 2023). Using these parameters instead of randomly selecting values ensures the results are reasonably representative of the final models.

The experiment to determine the optimal input sequence length was conducted similarly. Sequence lengths of 7, 14, and 28 were tested, corresponding to 1, 2, and 4 weeks of data, respectively. The performance across the three stocks was averaged to generalize the results.

The experiment to determine the optimal number of LSTM units and dropout rate required a warm start since there is no intuitive way to estimate reasonable ranges of these parameter values. Keras has a built-in hyperparameter search function that was used to find these warm starts. The function tested over a wide range of parameter values: 10-100 with a step size of 10 for LSTM units and 0 to 0.5 with a step size of 0.05 for dropout rate. This procedure did not incorporate walk-forward validation.

Finding these warm starts allowed for more rigorous manual testing of these parameters over a tighter range of values. Similar to the previous experiments, the number of units and dropout rate were varied with all other parameters held constant, and each model was evaluated using walk-forward validation.

The final experiment focused on determining the optimal number of LSTM stacks. The number of units determined from the previous experiment was used regardless of the number of stacks. Each stack had an equal number of units and was followed by a

dropout layer with a rate determined from the previous experiment. 1-10 stacks were tested.

GARCH Model Parameters

The GARCH parameters—autoregressive lag and moving average lag—were not optimized for this project; instead, both parameters were set to one. Since GARCH does not involve deep learning, the team did not want to spend a large amount of time optimizing these models. As such, GARCH predictions are displayed in the final output, but not evaluated for performance.

Evaluating LSTM Model Performance

A strategic design of this project was to initially train the models to predict the price of the first trading day of 2024 (1/2/24). While more data was available, the objective was to observe the model's performance over time, with daily refits to incorporate new information. The baseline model is to predict the prior day's price.

Several metrics were used to evaluate the LSTM models. First is the mean squared error on the normalized 2024 data. Second is the difference between the predicted and baseline mean absolute errors (MAE). Third is the mean directional error (MDE), which assesses if the model correctly predicted increases or decreases in closing price, regardless of magnitude.

Output Data for SlotMarketSQL Application

Once the LSTM and GARCH models were retrained on all available data, the forecasts and evaluation metrics were placed into the table that the web application references.

This table has information about the closing price, the next day's forecasted close price, the forecasted percentage change, and the prior 7 trading days' MAE and MDE. From the GARCH models, the table has current volatility, forecasted volatility, and forecasted percentage change.

The rationale for having the MAE and MDE be a 7-day average is for the day trader to better understand how the model has recently performed more holistically instead of just one day.

Building Slot-Filling Data

The slot-filling procedure required hand-labeled data so the SQL slots could be assigned to input tokens. To generate the dataset, a simple set of 16 types of reasonable questions was generated, and these questions served as the foundation of the test-set observations. ChatGPT 3.5 was used to create 20-25 permutations of each

question to serve as the training data. This generated ~400 training examples, which were subsequently hand-labeled.

There are a variety of potential output tokens, focusing on “select-[column name]” and “order-by-[column name]-[asc/desc]” as the potential labels, with O serving as the null/empty token. There is not a “from-[table name]” token as there is only one table. One clear limitation of this methodology is that filtering is not possible using a “where-[column name]-[sign]-[value]”. Still, selecting, no-selecting, and sorting are possible actions.

After manually labeling the ~400 observations, a find-and-replace methodology was used to generate similar but substantially different questions and to expand the vocabulary of the training corpus. There were thus 2,364 training observations. Similar observations were performed for the test set, where the original 16 questions were transformed into 64 observations.

Training Transformers

Extensive testing was undertaken to choose the best architecture and training structure. The chosen model architecture (shown in Snippets 1 and 3 in the Appendix) has 8,747,163 trainable parameters. Pre-trained models were not used for embeddings, transformers, etc. The traditional transformer encoder architecture was preserved.

SlotParser - From Tokens to SQL

The next step was translating the transformer’s output into a usable SQL query. Based on the setup of the slot tokens, the indicator value can be identified as either “select-”, “order-by-”, “-asc”, or “-desc”. String manipulations were performed to determine the appropriate column or sorting direction to use. To achieve this, a function named “SlotParser” was created, which returns both the properly formatted SQL query and the resulting data frame, as demonstrated in Snippet 4.

It’s important to ensure that no errors are possible based on the output of the transformer encoder. Therefore, some logical guards were put in place to avoid any impossible results. For example, if no tokens are detected in the output, the entire table is returned, sorted in alphabetical order by stock name. If a query sorts on a column that is not selected, it is added to the select statement to ensure the user can validate the query results.

Streamlit Application

A Streamlit application was created to allow the end user to interact with the workflow and to “package” this comprehensive model into a nice format. Streamlit is a low-code

way of hosting a web application, with the ability to publish to external users through the use of webhooks and GitHub repositories. The user interface was intentionally kept simple, focusing on the user's prompt with a simple button to execute the models in the background. Ideally, the data can get refreshed daily through a scheduled cron job, allowing for seamless updating of forecasts and LSTMs, without manual intervention. This automation is not in place, yet.

After a prompt is submitted, three outputs are returned. First is the requested table. Next are the original prompt, the outputted slot tokens, and the corresponding SQL query to make clear what commands were run and how to interpret the resulting data frame. Finally, as a convenient measure, the entire table is returned at the bottom.

Results

LSTM Architecture

Number of Epochs

Figures 3-5 in the Appendix show the average MSE from walk-forward validation for each stock for varying numbers of epochs. For each stock, the error flattens out around five epochs. Thus, all subsequent experiments used five epochs to avoid overfitting.

Input Sequence Length

Table 1 in the Appendix shows the results from varying the input sequence length. An input sequence length of 7 resulted in the lowest average MSE from walk-forward validation. Thus, all subsequent experiments used a sequence length of 7.

Number of LSTM Units and Dropout Rate

Keras' built-in hyperparameter search function was initially used to get parameter estimates for the number of LSTM units and dropout rate. Table 2 in the Appendix shows the results from these hyperparameter searches.

The results from these hyperparameter searches allowed for more precise ranges of values in subsequent experiments. These experiments were more rigorous, using walk-forward validation. Figure 6 shows the results from these manual hyperparameter tests.

The parameter combination that achieved the lowest average mean squared error is 90 LSTM units and a 15% dropout rate.

Number of LSTM Stacks

The number of LSTM stacks was tested using walk-forward validation. 6 LSTM stacks achieved the lowest average MSE. Figure 7 in the Appendix displays the results, which suggests that a smaller number of stacks is underfitting and a larger number is overfitting.

Final Architecture Summary

Figure 1 summarizes the final architecture of the models, based on the results from the experiments.

LSTM Model Evaluation

The models are evaluated by comparing the forecasted and actual outcomes.

Figure 8 in the Appendix illustrates an example of this comparison using Apple (AAPL).

Normalized MSE and MAE

The LSTMS were evaluated using mean squared error (MSE) on the normalized data. For each stock, the MSE was calculated across the entire 2024 evaluation period, with values ranging from 0.0005 - 0.264. Figures 2 in the appendix are histograms of the MSE distribution, with the height of the bars representing how many companies fell into each bucket. In Figure 2, the left graph shows the LSTM and the right shows the baseline (prior day price). The LSTM model only outperforms the baseline for 14 of the 497 stocks modeled.

The difference between the predicted mean absolute error (MAE) and the baseline mean absolute error was also considered. Negative numbers indicate the LSTM model performed better than the baseline model. Only 1 company had a positive difference in MAE. There are likely many reasons for the baseline outperforming the LSTM, which is discussed in the Lessons Learned section.

Figure 1: Architecture of final models

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 7, 1)]	0
lstm (LSTM)	(None, 7, 15)	1020
dropout (Dropout)	(None, 7, 15)	0
lstm_1 (LSTM)	(None, 7, 15)	1860
dropout_1 (Dropout)	(None, 7, 15)	0
lstm_2 (LSTM)	(None, 7, 15)	1860
dropout_2 (Dropout)	(None, 7, 15)	0
lstm_3 (LSTM)	(None, 7, 15)	1860
dropout_3 (Dropout)	(None, 7, 15)	0
lstm_4 (LSTM)	(None, 7, 15)	1860
dropout_4 (Dropout)	(None, 7, 15)	0
lstm_5 (LSTM)	(None, 15)	1860
dropout_5 (Dropout)	(None, 15)	0
dense (Dense)	(None, 1)	16
=====		
Total params: 10336 (40.38 KB)		
Trainable params: 10336 (40.38 KB)		
Non-trainable params: 0 (0.00 Byte)		

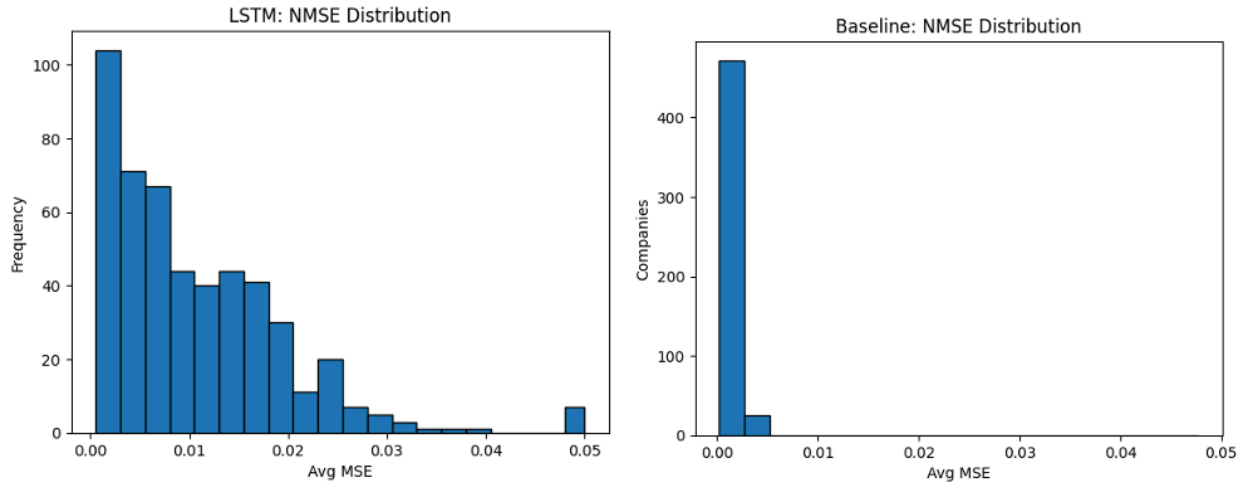


Figure 2: LSTM and Baseline NMSE distribution

Mean Directional Error

Lastly, the distribution of the mean directional error for the LSTM model over the entire 2024 period was considered. In the SlotMarketSQL Application, the output only considers the rolling 7 day MDE. For this section, the entire 2024 period was considered for a more holistic evaluation. A value of 1 indicates the prediction is always directionally aligned with the outcome and 0 indicates it is always directionally misaligned. The average MDE across all stocks was 49.1%, and Figure 9 in the appendix shows the corresponding histogram.

Slot-filling Accuracy

The pre-built PositionalEmbedding and TransformerEncoder classes, built by the HODL teaching team, were employed. This allowed for rapid testing since the code works immediately out of the box. While only training on a small corpus, the model achieved >99.6% accuracy in-sample, with test-set performance on the actual slot tokens achieving ~93%. The model is relatively small, takes less than 2 minutes to train, and is very quick in our web application.

Lessons Learned

LSTM Model

As discussed in the Results section, the baseline model generally outperforms the LSTM model. This is likely a result of several design choices made during development.

Data Used

No exogenous variables were used when training the LSTM models. Indeed, the only features are the closing prices of previous time periods. The stock market is very complicated and the closing price is likely influenced by the movement of other stocks,

how the market as a whole is moving, the amount of trading occurring for a given stock, etc.

Architecture Decisions

To simplify the training process, every stock shares the same model architecture, as determined by experimenting on a random sample of three stocks. This is a limitation because it is very likely each stock would perform better with a unique architecture and parameter set.

SlotMarketSQL Application

The methodology used performed well but has some limitations discussed below.

Manually Labeling Inputs to SQL

A large amount of manual data labeling was necessary. Luckily, ChatGPT was useful in coming up with various augmentations of inputs, but these still had to be manually labeled. A key limitation of this is, if a user enters a new query, the application is not able to fully handle the request. Ideally, the application would be more resilient and able to handle new, unique questions. An LLM may be a better choice by using system prompts and RAG to ensure quality responses.

Filtering SQL Queries

“WHERE” clauses were not considered due to their complexity in the manual labeling process. Indeed, filters could have multiple conditions with and/or links that add to the complexity of both labeling and properly identifying each part with the transformer.

Future Improvements

Across the LSTM models, GARCH models, and SlotMarketSQL Application, there are multiple areas of improvement.

LSTM Model

As discussed previously, there is room to include more data and customized architectures for each stock, which would improve the performance of the models. Doing so would require more computational power to find the best parameter sets for each stock.

GARCH Model

The GARCH models were not hypertuned or evaluated. Spending more time optimizing these models would significantly improve the volatility forecasts.

Transformer Model

The transformer model that translated user inputs to SQL queries did not use any pre-trained models and had limited samples. There were only 2,364 question examples; increasing this number could have increased test accuracy. Another option is to increase the vocabulary size to account for more types of questions.

Instead of creating the embeddings from scratch, there is also room to explore using pre-trained models such as BERT or GloVe, along with fine-tuning. This would help expand the vocabulary and get better representation of words.

Lastly, adding in a filtering (WHERE clause) function would produce a better user experience. This would take significant time with the current methodology, especially considering language complexities, such as understanding OR versus AND clauses.

Using LLMS

Another option is to overhaul the embeddings to SQL architecture and use LLMs. This method allows for a much larger vocabulary and would thus be able to handle more complex requests, such as multiple filtering conditions. The downside is the risk of hallucination and giving the user incorrect output. This could be mitigated by using system prompts and RAG to provide context to the LLM to guide responses. This could provide a meaningful increase in usability.

Works Cited

Charles T. Hemphill, John J. Godfrey, and George R. Doddington. 1990. The ATIS Spoken Language Systems Pilot Corpus. In *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27, 1990*.

Sample transformer code provided by HODL teaching staff
<https://www.dropbox.com/s/4rdgil1epnvgitf/HODL.py>

Gülmez, Burak. "Stock price prediction with optimized deep LSTM network with artificial rabbits optimization algorithm." *Expert Systems with Applications* 227 (2023): 120346.

Appendix

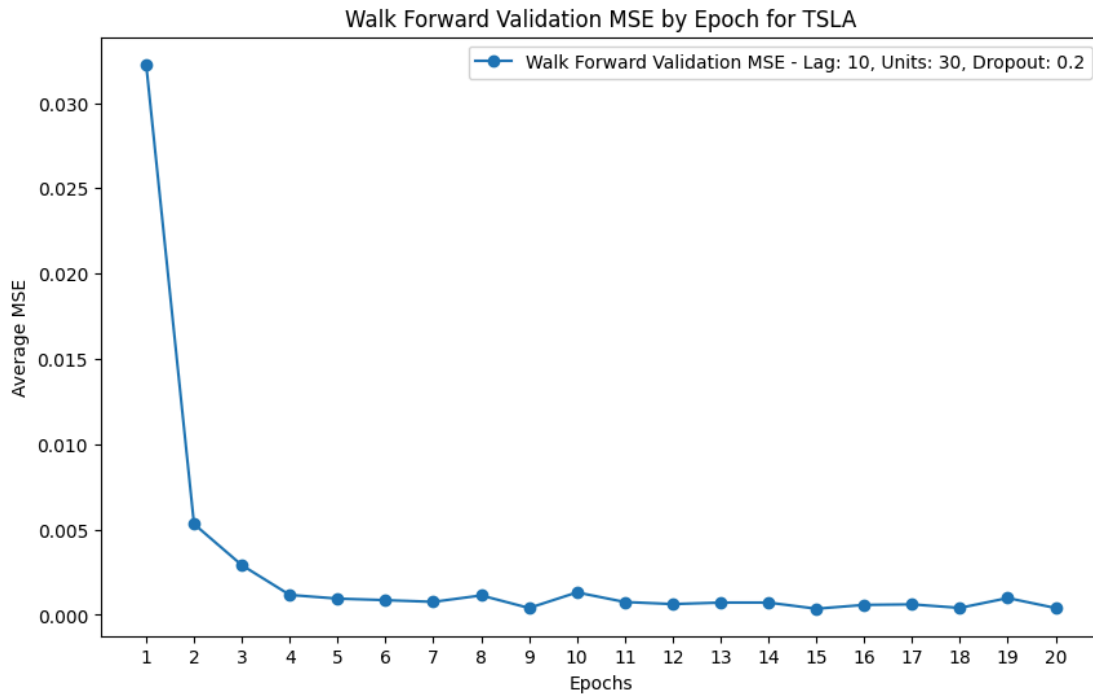


Figure 3: Average mean squared error from walk-forward validation across varying numbers of epochs for TSLA

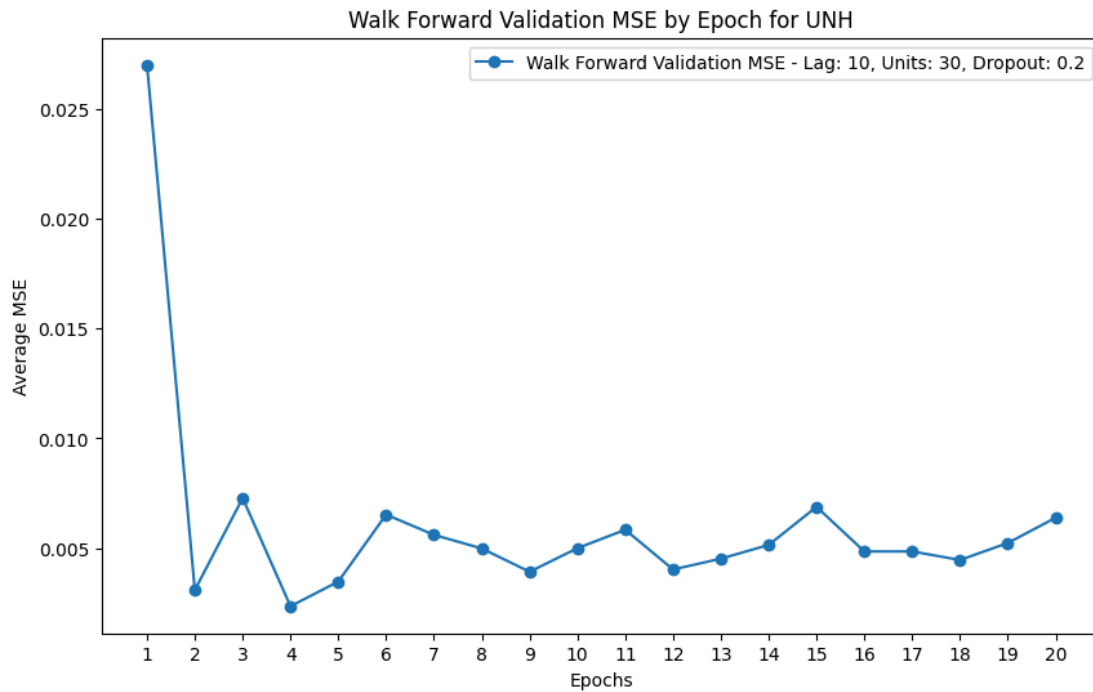


Figure 4: Average mean squared error from walk-forward validation across varying numbers of epochs for UNH

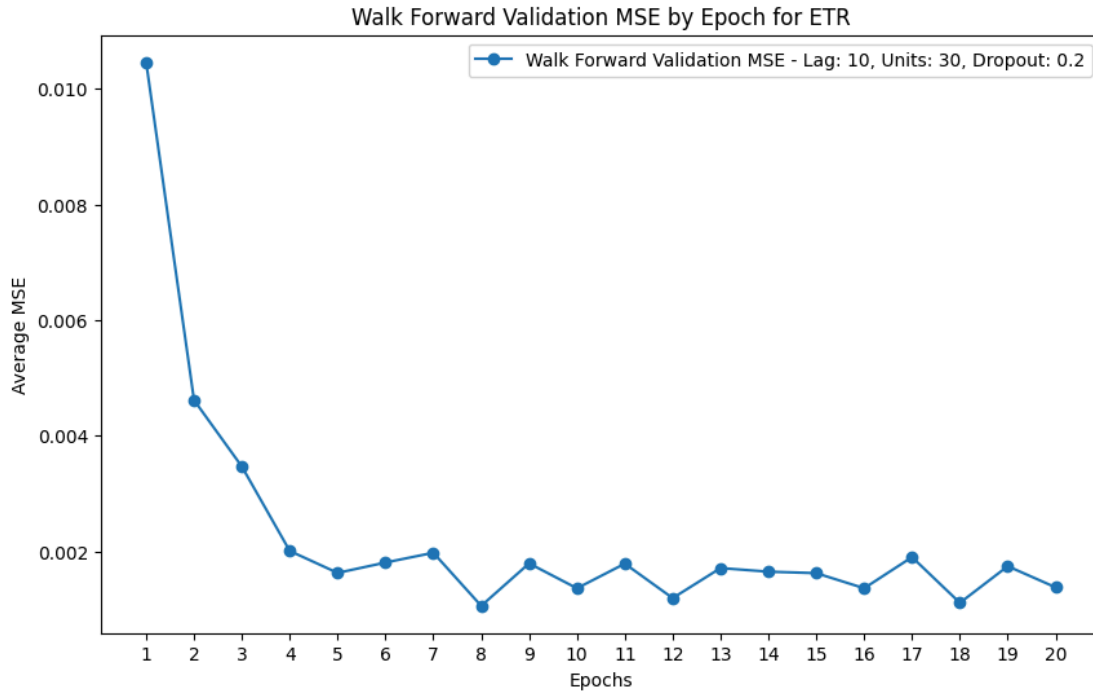


Figure 5: Average mean squared error from walk-forward validation across varying numbers of epochs for ETR

Table 1: Average mean squared error from walk-forward validation across varying input sequence lengths.

Input Sequence Length	Average MSE [10^{-3}]
7	2.57
14	3.88
28	3.84

Table 2: Number of LSTM units and dropout rate selected from Keras hyperparameter searches for each stock.

Stock	Number of LSTM Units	Dropout Rate
TSLA	90	10%

UNH	100	0%
ETR	50	15%

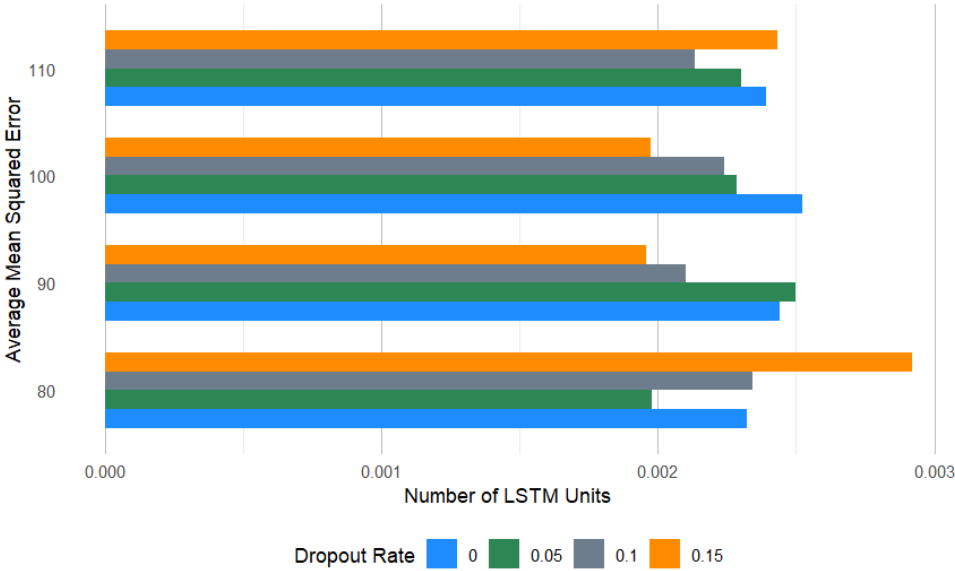


Figure 6: Average mean squared error from walk-forward validation across varying numbers of LSTM units and dropout rates.

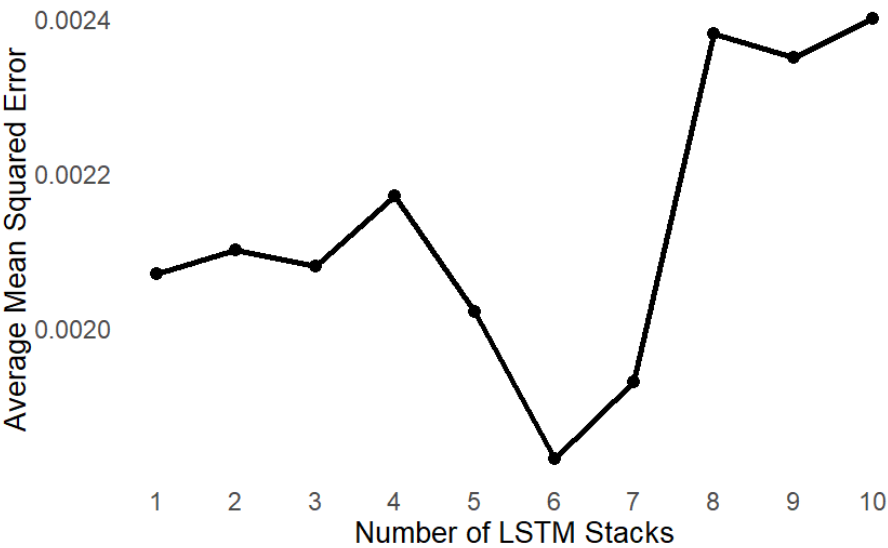


Figure 7: Average mean squared error from walk-forward validation across varying numbers of LSTM stacks.

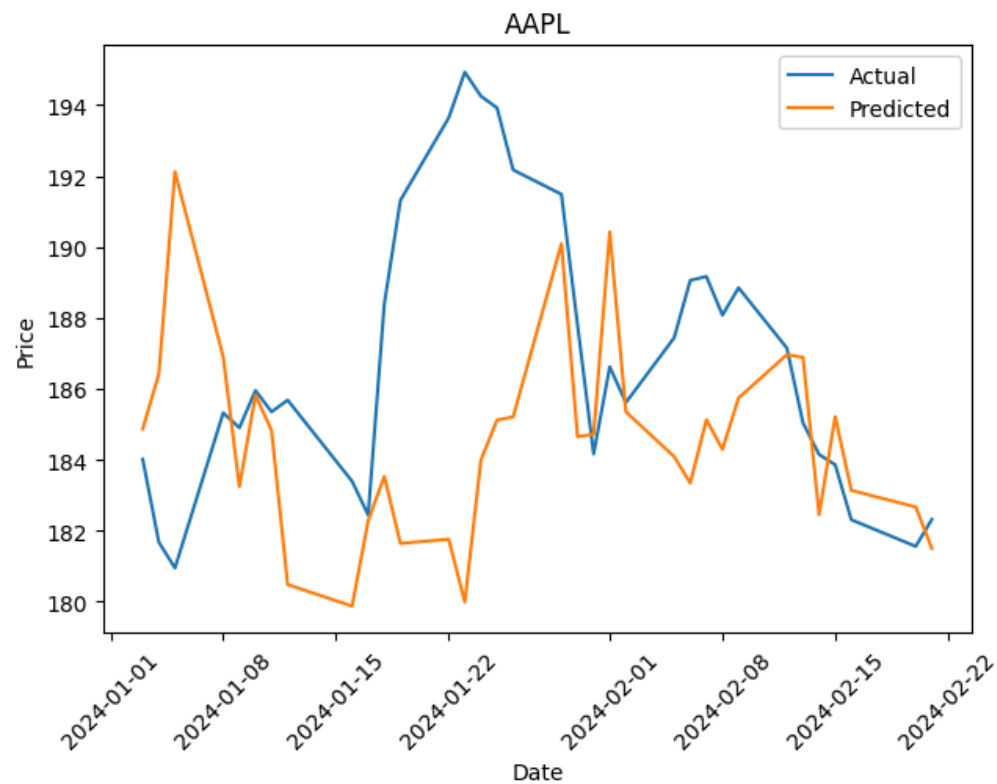


Figure 8: Apple's Actual vs Predicted Prices

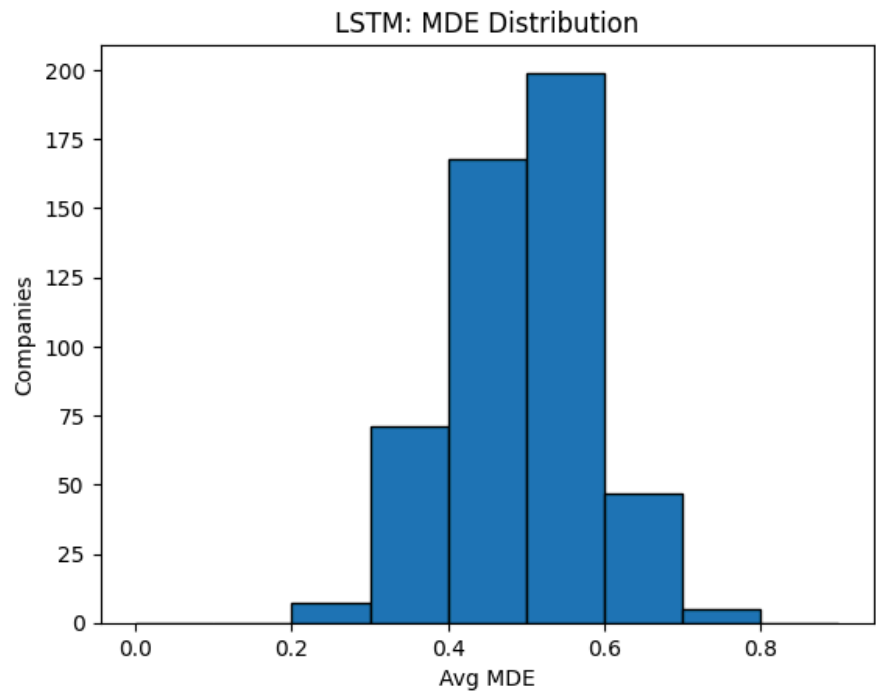



Figure 9: LSTM MDE distribution



HODL-Project - NLP_to_SQL.py

```

13 MAX_QUERY_LENGTH = 50 #size of input
14 EMBED_DIM = 512 #dimension of embeddings
15 DENSE_DIM = 128
16 NUM_HEADS = 8 #number of multi-attention heads
17 DENSE_UNITS = 128 #num nodes in hidden layer
18 BATCH_SIZE = 64 #batch size for training transformer
19 EPOCHS = 10 #epochs for training transformer

```

Snippet 1: Hyperparameters for transformer model


HODL-Project - NLP_to_SQL.py

```

31 # CREATE VECTORIZER (QUERY & SLOTS)
32 vectorize_query_text = keras.layers.TextVectorization(
33     max_tokens=None, #no maximum vocabulary
34     output_sequence_length=MAX_QUERY_LENGTH, #pad or truncate output to value
35     output_mode="int", #vector has index of vocabulary
36     standardize="lower_and_strip_punctuation", #convert input to lowercase and rmv punctuation
37     split="whitespace", #split values based on whitespace
38     ngrams=1 #only look at whole words
39 )
40 vectorize_slot_text = keras.layers.TextVectorization(
41     max_tokens=None, #no maximum vocabulary
42     output_sequence_length=MAX_QUERY_LENGTH,
43     output_mode="int", #vector has index of vocabulary
44     standardize="lower", #convert input to lowercase [can't do punctuation b/c of dashes]
45     split="whitespace", #split values based on whitespace
46     ngrams=1 #only look at whole words
47 )
48
49 # CREATE VOCABULARY AND VECTORIZED TRAINING DATA
50 vectorize_query_text.adapt(train_query) #build vocabulary
51 query_train = vectorize_query_text(train_query) #vectorized training queries
52 query_test = vectorize_query_text(test_query) #vectorized testing queries
53 QUERY_VOCAB_SIZE = vectorize_query_text.vocabulary_size() #total vocabulary of queries
54
55 vectorize_slot_text.adapt(train_slotfilling) #build slot vocabulary
56 slots_train = vectorize_slot_text(train_slotfilling) #vectorized training slots
57 slots_test = vectorize_slot_text(test_slotfilling) #vectorized testing slots
58 SLOT_VOCAB_SIZE = vectorize_slot_text.vocabulary_size() #total vocabulary of slots

```

Snippet 2: Tokenization Layers

```

HODL-Project - NLP_to_SQL.py

60 # BUILD KERAS MODEL
61 inputs = keras.Input(shape=(MAX_QUERY_LENGTH,))
62 embedding = PositionalEmbedding(MAX_QUERY_LENGTH,
63                                QUERY_VOCAB_SIZE,
64                                EMBED_DIM)
65 x = embedding(inputs)
66 encoder_out = TransformerEncoder(EMBED_DIM,
67                                  DENSE_DIM,
68                                  NUM_HEADS)(x)
69 x = keras.layers.Dense(DENSE_UNITS, activation="relu", name="Dense_Layer")(encoder_out)
70 x = keras.layers.Dropout(0.25, name="Dropout_Layer")(x)
71 outputs = keras.layers.Dense(SLOT_VOCAB_SIZE, activation="softmax", name="Softmax_Layer")(x)
72
73 model = keras.Model(inputs, outputs)
74 print()
75 print(model.summary())
76 print()
77
78 # TRAIN KERAS MODEL
79 model.compile(optimizer="adam",
80               loss="sparse_categorical_crossentropy",
81               metrics=["sparse_categorical_accuracy"])
82 history = model.fit(query_train, slots_train,
83                     batch_size=BATCH_SIZE,
84                     epochs=EPOCHS)
85
86 # OUT-OF-SAMPLE TESTING
87 model.evaluate(query_test, slots_test)
88
89 # SAVE MODEL
90 filename = 'sql_transformer.keras'
91 model.save(filename)
92 ZipFile('model_save.zip', mode='w').write(filename)

```

Snippet 3: Building keras model

```

HODL-Project - useful_functions.py

14 def SlotParser(slot_filling, prompt, stock_data):
15     slot_filling = slot_filling.strip() #strip any whitespace from slot_filling return
16     # initialize aspects of the SQL query
17     slots = {'select': [],
18             'order': [],
19             'limit': None}
20
21     # for word (token) in prompt
22     for word in prompt.split(" "):
23         if word.isnumeric(): #if we found a number
24             slots['limit'] = int(word) #assume number relates to LIMIT (# of rows to display)
25
26     # for slot (token) in output
27     for token in slot_filling.split(" "):
28         if (token != "o") and (token != "0"): #if not a null slot
29             if 'select-' in token: #if a select slot
30                 if token not in slots['select']: #if we haven't already added it
31                     slots['select'].append(token) #then include in SELECT statement
32             elif 'order-by' in token: #if an order-by slot
33                 if token not in slots['order']: #if we haven't already added it
34                     slots['order'].append(token) #then include in ORDER BY statement
35
36     if len(slots['select']) == 0: #if we haven't selected a column
37         columns = list(stock_data.columns) #assume all
38     else: #if we have selected >= 1 column
39         columns = [x.split("-")[1] for x in slots['select']] #access column names, format them
40
41     if len(slots['order']) == 0: #if we haven't ordered a column
42         if 'stock' not in columns:
43             columns.insert(0, 'stock') #insert at beginning
44         order_cols = ['stock'] #assume we order by stock
45         order_ascending = [True] #assum ascending order
46     else: #if there is order statement
47         order_cols = [x.split("-")[2] for x in slots['order']] #format
48         order_ascending = [True if x.split("-")[-1] == 'asc' else False for x in slots['order']] #format
49
50     if slots['limit'] == None: #if we don't have a number
51         limit = len(stock_data) #assume we want all rows
52     else: #if there is a limit number
53         limit = int(slots['limit']) #make sure to return that many rows
54
55     for col in order_cols: #for every ordering column
56         if col not in columns: #check if it is an accessible column
57             columns.append(col) #if not, add it
58
59     pandas_query = stock_data[columns] #use only selected columns
60     pandas_query = pandas_query.sort_values(by=order_cols, #columns to sort
61                                           ascending=order_ascending, #boolean list of asc/desc
62                                           ignore_index=True)
63     pandas_query = pandas_query.head(limit) #LIMIT statement
64
65     all_data = stock_data.sort_values(by=order_cols,
66                                     ascending=order_ascending,
67                                     ignore_index=True) #include just in case
68     # print(pandas_query) #print the dataframe
69
70     #formatting SQL statements
71     SELECT = f"SELECT {' '.join(columns)}"
72     FROM = "FROM stock_data"
73     ORDER_BY = f"ORDER BY {' '.join((str(x.split('-')[2])+' '+str(x.split('-')[-1].upper())) for x in slots['order']))"
74     LIMIT = f"LIMIT {slots['limit']}"
75
76     #checking for errors
77     if len(slots['select']) == 0: #empty slots
78         SQL_QUERY = "SELECT *\n FROM stock_data"
79     elif (len(slots['order']) == 0) and (slots['limit'] == None): #no sort or limit statement
80         SQL_QUERY = f"{SELECT}\n{FROM}"
81     elif len(slots['order']) == 0: #no order clause
82         SQL_QUERY = f"{SELECT}\n{FROM}\n{LIMIT}"
83     elif slots['limit'] == None: #no limit clause
84         SQL_QUERY = f"{SELECT}\n{FROM}\n{ORDER_BY}"
85     else:
86         SQL_QUERY = f"{SELECT}\n{FROM}\n{ORDER_BY}\n{LIMIT}"
87
88     print(SQL_QUERY) #print SQL statement
89
90     return pandas_query, SQL_QUERY, all_data

```

Snippet 4: Code for SlotParser function