

## Appendix: Generative AI Assistance (ChatGPT)

Prompt:

How might I use VADER to then perform sentiment analysis on my data?

Query driven by visit to source: <https://realpython.com/python-nltk-sentiment-analysis/>

Response:

To perform sentiment analysis using VADER (Valence Aware Dictionary and sEntiment Reasoner) on your data in a Pandas DataFrame, you'll need to follow these steps:

### Step 1: Install VADER

VADER is included in the nltk (Natural Language Toolkit) package. If you haven't installed NLTK, you can do so via pip. You also need to download the VADER lexicon.

```
bash
```

Copy code

```
pip install nltk
```

In your Python script, download the VADER lexicon:

```
python
```

Copy code

```
import nltk
```

```
nltk.download('vader_lexicon')
```

### Step 2: Import and Initialize VADER Sentiment Analyzer

```
python
```

Copy code

```
from nltk.sentiment import SentimentIntensityAnalyzer
```

```
sia = SentimentIntensityAnalyzer()
```

### Step 3: Apply Sentiment Analysis to DataFrame

Assuming you have a DataFrame df with a text column (e.g., 'Text'), you can apply VADER to perform sentiment analysis:

python

Copy code

```
import pandas as pd
```

Example DataFrame

```
data = {'Text': ['I love sunny days in New York.', 'The movie was terrible and boring.']}
```

```
df = pd.DataFrame(data)
```

Function to get sentiment

```
def get_sentiment(text):  
    return sia.polarity_scores(text)
```

Apply the function to the DataFrame

```
df['Sentiment'] = df['Text'].apply(get_sentiment)
```

Display the DataFrame

```
print(df)
```

This script will create a new column in your DataFrame called 'Sentiment', where each row contains a dictionary with scores for each of the four categories in VADER: 'neg' for negative, 'neu' for neutral, 'pos' for positive, and 'compound' for the aggregated score.

Prompt:

If I have two date values for a start and end date, how would I get all of the values in between them? For example, if I have 2015-11-16 to 2015-11-22?

Response:

```
from datetime import datetime, timedelta
```

```
# Start and end dates
```

```
start_date = datetime.strptime("2015-11-16", "%Y-%m-%d")
end_date = datetime.strptime("2015-11-22", "%Y-%m-%d")

# Generate all dates between start and end date

date_generated = [start_date + timedelta(days=x) for x in range(0, (end_date-start_date).days + 1)]

# Convert to strings if needed

date_strings = [date.strftime("%Y-%m-%d") for date in date_generated]

date_strings
```

Prompt:

What would be a simple Transformer model to implement for analysis of stock data and sentiment data?  
All of the data is already numeric in a dataframe.

Response:

For a simple Transformer model applied to stock and sentiment data, you can start with a basic architecture that includes an embedding layer (if necessary), Transformer encoder layers, and a final dense layer for prediction. Since your data is numeric and structured, you might skip the embedding layer or use it to project numeric features into a higher-dimensional space, depending on the complexity of your task.

Here's a basic outline of what such a Transformer model could look like using TensorFlow and Keras.  
This example assumes you're working on a regression task, such as predicting future stock prices:

Step 1: Import Necessary Libraries

python

Copy code

```
import numpy as np
```

```
import tensorflow as tf
```

```
from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, Dropout

from tensorflow.keras.layers import MultiHeadAttention, LayerNormalization

from tensorflow.keras.optimizers import Adam
```

## Step 2: Define Transformer Block

A Transformer block consists of multi-head self-attention and position-wise feed-forward networks, along with layer normalization.

python

Copy code

```
class TransformerEncoder(tf.keras.layers.Layer):

    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):

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

        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)

        self.ffn = tf.keras.Sequential(

            [Dense(dense_dim, activation="relu"), Dense(embed_dim),]

        )

        self.layernorm1 = LayerNormalization(epsilon=1e-6)

        self.layernorm2 = LayerNormalization(epsilon=1e-6)

        self.dropout1 = Dropout(0.1)

        self.dropout2 = Dropout(0.1)

    def call(self, inputs, training):

        attn_output = self.att(inputs, inputs)
```

```

attn_output = self.dropout1(attn_output, training=training)

out1 = self.layernorm1(inputs + attn_output)

ffn_output = self.ffn(out1)

ffn_output = self.dropout2(ffn_output, training=training)

return self.layernorm2(out1 + ffn_output)

```

### Step 3: Build the Model

Assemble the Transformer model. This example builds a model that takes sequences of stock and sentiment data as input.

python

Copy code

```

def build_model(input_shape, num_heads, embed_dim, dense_dim):

    inputs = Input(shape=input_shape)

    x = TransformerEncoder(embed_dim, dense_dim, num_heads)(inputs)

    x = GlobalAveragePooling1D()(x)

    x = Dropout(0.1)(x)

    outputs = Dense(1)(x)

    model = Model(inputs, outputs)

    return model

```

### Step 4: Compile and Train the Model

Compile and train your model. Adjust the learning rate, batch size, and number of epochs as necessary.

python

Copy code

```
input_shape = (sequence_length, num_features) # Adjust based on your data

embed_dim = 32 # Embedding size for each token

num_heads = 2 # Number of attention heads

dense_dim = 32 # Hidden layer size in feed forward network inside transformer


model = build_model(input_shape, num_heads, embed_dim, dense_dim)

model.compile(optimizer=Adam(learning_rate=1e-4), loss='mean_squared_error')


# Assuming X_train and y_train are prepared

history = model.fit(X_train, y_train, batch_size=32, epochs=100, validation_split=0.2)
```

#### Notes

**Input Shape:** Adjust `input_shape` to match your data. It should be `(sequence_length, num_features)`, where `sequence_length` is the number of time steps you're including in each sample, and `num_features` includes both stock and sentiment features.

**Feature Engineering:** Even though the Transformer model can capture complex dependencies, thoughtful feature engineering can significantly impact model performance. Consider including lagged stock features, moving averages, and sentiment scores as part of your feature set.

**Hyperparameters:** The model's performance can be sensitive to the choice of hyperparameters, including the number of attention heads, the dimensionality of the Transformer, and the learning rate. Experiment with different configurations to find the best setup for your specific dataset.

This simple Transformer model provides a starting point for analyzing stock and sentiment data.

Depending on your task's complexity and dataset size, you may need to adjust the model architecture, depth, and training parameters.