Sentiment Analysis with Python – A Beginner's Guide

· 20 min read

 34091 total views

Last Updated on July 7, 2020

Sentiment analysis in finance has become commonplace. In many cases, it has become ineffective as many market players understand it and have one-upped this technique.

That said, just like machine learning or basic statistical analysis, sentiment analysis is just a tool. It is how we use it that determines its effectiveness.

Here are the general steps to learn sentiment analysis for finance:

- **Understand what Sentiment Analysis is**
- **Understand how it can be effectively used in finance**
- **Learn data collection and text processing**
- **Learn to run sentiment analysis**
- **Learn how to use the analysis output for finance**

Table of contents

Let's first understand why we need sentiment analysis for finance, or more specifically, trading.

Next, we will demonstrate a project that uses Python to extract and analyse article headlines to predict Tesla's stock prices.

Let's go!

What is Sentiment Analysis?

Official Definition (from Wikipedia):

Sentiment analysis (also known as opinion mining or emotion AI) refers to the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information.

In simple English:

We use computers to extract meanings behind texts, images and other data.

Why do we need Sentiment Analysis?

Why can't humans just read the texts? Why do we need a machine to do it for us?

Reasons for using sentiment analysis:

- Machines can read much faster (maybe a million times faster) than humans
- Machines can read in many languages
- Machines can derive meaning from text in a standardised manner (humans are subjective)
- Machines can store insights from texts in a convenient way for further processing

There are of course downsides to sentiment analysis. Machines are not able to accurately derive meaning from texts (but they are getting better). Slangs, typos, contextual meaning, sarcasm still poses difficulties.

What is sentiment analysis for finance?

Financial sentiment analysis is used to extract insights from news, social media, financial reports and alternative data for investment, trading, risk management, operations in financial institutions, and basically anything finance related.

We will focus on trading and investments in this article.

How is sentiment analysis used for trading?

Here are a few ways:

- Read a news article or tweet fast and fire a trade instantly
- Read large amount of financial reports and output insights
- Gather insights from the crowds by analysing social media, web forums, news and analysts' reports

Read a news article or tweet fast and fire a trade instantly

Let's say that Country A's leader decided to make a trade deal with another out of the blue.

In the best case scenario, a human might take 2 seconds to read that piece of news (if he or his team is awake) and another 3 seconds to fire an appropriate trade (if he is fast and is already on his trading desk).

A machine would take less than 0.1 seconds to read the new and fire the trade. Plus, the machine doesn't sleep and can monitor the news from not only Country A, but all countries around the way.

Read large amount of financial reports and output insights

A machine can read 1000 annual 10-K financial reports (in any language) in the time you take to read the first 10 pages of one report.

That said, machines aren't that great in deriving insights from such large unstructured text data.

There is a large variance in output. The machine might get it right on average when you combine insights from 1000 stocks, but for an individual stock, it will get it wrong most of the time.

Each insight can have a big variance (i.e. they might miss the mark some of the time)

Thus, the value here might not be to derive insights for one stock. It is to derive insights from thousands of stocks, traded in the same portfolio in a statistical manner.

We will get an average prediction when combining many stock insights.

The variance in each stock insight will balance out when we combine it with thousands of other stocks. Hence, we will get an average prediction for our portfolio of hundreds or thousands of stocks.

This is similar to the idea in central limit theorem.

Once we get our average prediction and standard deviation figures, we can then input that into a sizing algorithm to determine how much we should trade for each stock and how to allocate capital for the portfolio to maximise long term reward-to-risk ratio. But this is a story for another day.

But, do note that if your sentiment analysis of the financial reports is so bad that the mean of your insights is inaccurate, then you will not be profitable anyways.

Gather insights from the crowds by analysing social media, web forums, news and analysts' reports

This is touchy. Sentiment analysis of social media posts were hyped up a few years ago. The effectiveness of these analysis remains debatable.

That said, we can increase the effectiveness of these insights by complementing them with other analysis, or to sandbox them by hedging away the variables we can't control.

To read more on sandboxing: How to use Hedging as a Trading Strategy

For instance, if we are thinking of investing in Slack but are worried that Microsoft Teams will make Slack obsolete.

We can go to tech forums and check the amount and sentiment of the comments there about Slack vs that of Microsoft Teams.

When we do a pairing using the same information source, the results are generally more accurate as most unwanted variables will be hedged away.

On the other hand, if we have just taken forum comments on Slack and try to assign a score of how positive or negative it is, the results will be subjective.

It will contain variables like, the accuracy of the sentiment analysis library, the methodology in text processing, noise and low quality data etc.

Teams is catching up to Slack! – Credits: Google Trends

The lazy way is to check the search traffic for Slack vs Teams on Google Trends.

Now that we've covered the theory, let's get our hands dirty!

» Hello! This article doesn't cover live trading, check out this guide if you want to learn how to run a live algorithmic trading: Alpaca Trading API Guide – A Step-by-step Guide

How to predict stock prices with news and article headlines?

Here are the steps to run our sentiment analysis project:

1. **Collate article headlines and dates**
2. **Import and clean the data (text processing)**
3. **Run sentiment analysis and create a score index**
4. **Correlate lagged score index against prices**

This is the basic overview. Of course, the effectiveness of our analysis lies in the subtle details of the process.

Mega Project: Predicting Tesla stock prices with Seeking Alpha's article headlines with Python

We will be checking if Seeking Alpha's headlines have any predictive power for Tesla's stock price movements.

This will be done using the above 4-Step process with Python.

We will conduct a very basic level of analysis to keep things simple.

1. Collate article headlines

Let's download a web scrapping package called BeautifulSo... Just kidding!

This is not a web scrapping article and I don't want to bloat it. We will scrape the headline by hand!

Here are the steps for collating headlines:

1. Go to SeekingAlpha.com, search for TSLA and scroll for more headlines
2. Copy and paste the page onto Excel
3. Remove unwanted data

Step 1: Go to SeekingAlpha.com, search for TSLA and scroll for more headlines

Go to SeekingAlpha.com and search for TSLA (Tesla's ticker symbol in the search bar at the top of the page.

You will see a page like this. What we want is the headline under the Analysis section.

Before we copy that, keep scrolling down to load more headlines.

In my analysis, I scrolled down till the early 2018 articles appeared.

Step 2: Copy and paste the page onto Excel

Next, ctrl-A the page. Yes, you read that right. We are going old school.

Open your Excel, then ctrl-C. You should see something like this

Step 3: Remove unwanted data

Delete all the unwanted rows. We want to keep the "Analysis" headlines (not the "News" headlines) and the corresponding dates.

Delete all rows above the first headline.

Your first row should be the first "Analysis" headline. Ignore the thumbnail pictures, they will be gone later when we save the file as a CSV.

Delete all rows below the date of the last headline. In other words, delete all rows starting with the text "News" in bold.

You can search for "News" and check "Match entire cell contents" to find that row.

Now, save that file as a CSV. This will remove all the thumbnail graphics.

2. Import and clean the data (text processing)

We will use Python and Jupyter Notebook for this. Python is a programming language and Jupyter Notebook is the "software" that we code in. The technical term is IDE (Integrated development environment).

You can use other IDEs, but I suggest using Jupyter Notebook if you are new to this.

For those who are new, you can check out these guides on how to install Python and Jupyter Notebook on your computer using Anaconda: Hackernoon Guide, Anaconda Docs Guide

Here are the steps for this section:

1. Import your CSV to your Jupyter Notebook
2. Clean our data

Step 1: Import your CSV to your Jupyter Notebook

We'll use the pd.read_csv() method in Pandas to pull our CSV in.

Pandas is a Python library for the purpose of data science. You can install it by following: https://pypi.org/project/pandas/ or https://anaconda.org/anaconda/pandas

```
import pandas as pd # import the Pandas Library

df1 = pd.read_csv("tesla-headlines-sa.csv", encoding='windows-1250', header=None)
df1.columns = ['Title', 'Date']
df1
```

If you are too lazy to copy and paste headlines from the SeekingAlpha website, you can use our dataset.

Download the our entire code + data folder from our Github repository: Sentiment-Analysis-1-TSLA-Headlines. The CSV file is called "tsla-headlines-sa.csv".

Make sure that your CSV file is in the same folder as where your code is saved if you are running my code.

Importing our CSV

Running pd.read_csv() will give us a dataframe with 2 columns. We've titled them "Title" and "Date". We've added an encoding input to fix the character formatting issue.

Step 2: Clean our data

The next step is to clean our data.

Our "Title" data is already clean enough to be used for our sentiment analysis library, so we shall leave it as it is.

Our "Date" Data needs work though. Here are the steps to clean the date data

1. Determine our end goal
2. Clean the dates
3. Convert cleaned date to datetime format
4. Clean and convert the entire dataframe

1. Determine our end goal

Our Date data is in text (i.e. string) format. We want to change it to a datetime format so that it is easier to run our analysis along with our stock price data later.

2. Clean the date

Before we can modify the date using code, we need to briefly look through the dataset to have a sense of the format of the data.

I've briefly scanned through the data, and spotted 4 variations.

Variation 1:

Variation 1

Variation 1 doesn't contain a day or date. It says "Yesterday". Only the first row has this format.

Thus, I change this date via hard coding since it is inefficient to create a systematic code when it will only be used once.

```
df1['Date'][0] = 'Dec. 9'
df1.head() # display the first 5 rows
```

I change the format to a text similar to the other rows. Hence, when I modify the other rows using code, the first row will be modified too.

The first row is changed.

Variation 2

Variation 2

Variation 2 consists of the day, date but it doesn't have a year.

SeekingAlpha doesn't include the year if the article is published in the same year as the current year.

We are not interested in the day. We just want the date and year.

Here, we need to extract the date and add in the current year.

```
import re # import Regular expression library

print(df1['Date'][1]) # display original

match = re.search(r'\w{3}\.\s\d{1,2}', df1['Date'][1])

modifiedDate = match[0] + ", 2019"

print(modifiedDate) # display modified
```


To do this, we first import the Regular Expressions library (AKA re AKA Regex library) to help us with string manipulation.

We look for dates with the format "\w{3}.\s\d{1,2}".

- \w{3} looks for 3 letters
- \. looks for a period symbol
- \s looks for a space
- \d{1,2} looks for 1 or 2 digits

This format fits our variation 2 data, which looks like "Dec. 6". All other texts are ignored.

Here is a character cheat sheet for reference.

After we found our date, we add the year to it.

If you are wondering, "I'm new to Python, how do I know what code to type?"

The answer is... you Google it.

The beauty about coding is that you are building on top of other people's knowledge and work.

You might want to learn some bare minimum basics. Then whatever problem you want to solve, Google it, copy other people's code, modify it, make mistakes, learn and repeat.

After a while, you will be faster at this and can solve problems more effectively (still with the help of Google).

A good programmer is not someone who can spin up effective code out of thin air (though those people do exist). A good programmer knows what he doesn't know, what his tools can achieve (even though he might not know how to do it) and how to find answers.

We add ", 2019" instead of "2019" to match variation 3.

Variation 3

Variation 3

Variation 3 is simply variation 2 plus the year.

```
import re # import Regular expression library

print(df1['Date'][1000]) # display original

match = re.search(r'\w{3}\.\s\d{1,2}\,\s\d{4}', df1['Date'][1000])

print(match[0]) # display modified
```

The code is similar to variation 2. We added "\d{4}" in the re.search to grab the year.

Variation 4

Variation 4 is specific to the month of May.

Variation 4

All months except May have a period symbol after it. "Jan.", "Feb." etc. The period exists to indicate the spelling of the month is truncated.

The month of May doesn't need this.

Thus, in our Regex code, we do not need to include a period symbol.

Note that to see all the data in your dataframe, you can use the following code:

```
pd.set_option('display.max_rows', df1.shape[0]+1)
df1
```

We have 2 code for variation 4. One for the dates with year, one for dates without.

```
import re # import Regular expression library

print(df1['Date'][200]) # display original (without year)
match = re.search(r'\w{3}\s\d{1,2}', df1['Date'][200])
modifiedDate = match[0] + ", 2019"
print(modifiedDate) # display modified

print(df1['Date'][850]) # display original (with year)
match = re.search(r'\w{3}\s\d{1,2}\,\s\d{4}', df1['Date'][850])
print(match[0]) # display modified
```


3. Convert cleaned date to datetime format

Now that we have all the dates in either "MMM. DD, YYYY" or "May DD, YYYY" format, it is time to convert these to datetime format.

```
from datetime import datetime

newDate1 = datetime.strptime('Dec. 6, 2019', '%b. %d, %Y').date()
print(newDate1)

newDate2 = datetime.strptime('May 17, 2018', '%b %d, %Y').date()
print(newDate2)
```

Import the datetime library. This library helps us with datetime formatting.

Use the datetime.strptime() method to convert date to time. The first input is our date, the second input is the format of our date.

The symbols " %b. %d, %Y" represent the date formats.

- %b looks for the months' 3 character shortname

- . looks for a period symbol
- %d looks for the day of the month as a number
- %Y looks for the year as a 4 digit number

You can learn more about datetime.strptime() here.

4. Clean and convert the entire dataframe

Now that we've covered how to clean the 4 variations and convert the date to the datetime format, lets' run a loop to clean the entire "Date" column.

```
from datetime import datetime
import re

newDateList = [] # create a list to store the cleaned dates

for dateOfArticles in df1['Date']: # loop every row in the "Date" column
    match =
re.search(r'\w{3}\.\s\d{1,2}\,\s\d{4}|May\s\d{1,2}\,\s\d{4}|\w{3}\.\s\d{1,2}|May\
s\d{1,2}',
                      dateOfArticles)

    if re.search(r'\w{3}\.\s\d{1,2}\,\s\d{4}|\w{3}\s\d{1,2}\,\s\d{4}',match[0]):
        fulldate = match[0] # don't append year to string
    else:
        fulldate = match[0] + ", 2019" # append year to string

    for fmt in ('%b. %d, %Y', '%b %d, %Y'):
        try:
            newDate = datetime.strptime(fulldate, fmt).date()
            break # if format is correct, don't test any other formats
        except ValueError:
            pass

    newDateList.append(newDate) # add new date to the list

if(len(newDateList) != df1.shape[0]):
    print("Error: Rows don't match")
else:
    df1['New Date'] = newDateList # add the list to our original dataframe

df1
```


Wow that's a handful of code. Don't worry we will break it down:

```
for dateOfArticles in df1['Date']: # loop every row in the "Date" column
    match =
re.search(r'\w{3}\.\s\d{1,2}\,\s\d{4}|May\s\d{1,2}\,\s\d{4}|\w{3}\.\s\d{1,2}|May\
s\d{1,2}',
                        dateOfArticles)
```

Here we loop through every row and look for any of the 4 date string variations.

Note that the "|" symbol represents "or". It allows us to look for one variation or another.


```
if re.search(r'\w{3}\.\s\d{1,2}\,\s\d{4}|\w{3}\s\d{1,2}\,\s\d{4}',match[0]):
        fulldate = match[0] # don't append year to string
    else:
        fulldate = match[0] + ", 2019" # append year to string
```

Once we found the variation, we check if it contains the year.

If yes, don't add a year to the string. If no, add the appropriate year to the end of the string.

```
for fmt in ('%b. %d, %Y', '%b %d, %Y'):
        try:
            newDate = datetime.strptime(fulldate, fmt).date()
            break # if format is correct, don't test any other formats
        except ValueError:
            pass

newDateList.append(newDate) # add new date to the list
```

Next, we convert the "Date" data from string to datetime format.

Our dates have 2 possible formats now, one with a period symbol and one without. We will check for both.

Once done, add the new date data to a list.

```
if(len(newDateList) != df1.shape[0]):
    print("Error: Rows don't match")
else:
    df1['New Date'] = newDateList # add the list to our original dataframe
```


Check if the number of rows of the list match with original dataframe.

If yes, add the list as new column to our original dataframe.

We have finally gotten our “Date” data fixed!

3. Run sentiment analysis and create a score index

The next part is to send our headlines into a sentiment analyser to churn out a score.

We can build our own sentiment analyser model. A simple one can be something that is trained using supervised machine learning.

The training data can be historical financial headlines. The individual words, phrases, or entire headlines in this data set will be labelled with a sentiment score. E.g. 1 could be extremely positive, 0 is neutral and -1 is extremely negative. This is known as lexicon-based sentiment analysis.

You can think of a lexicon as a list of words, punctuation, phases, emojis etc.

We can then use this trained model to evaluate the sentient score for future headlines.

All of this model building stuff sounds fun but... we won't be doing that in this article. I will write another article dedicated to sentiment analysis model building.

In this article, we will use pre-trained models that are built by others.

Introducing... Vader

What is VADER Sentiment Analyzer

VADER is a sentiment analyser that is trained using social media and news data using a lexicon-based approach. This means that it looks at words, punctuation, phases, emojis etc and rates them as positive or negative.

VADER stands for "Valence Aware Dictionary and sEntiment Reasoner". You can learn more about it here and here.

Using VADER to analyse TSLA Headlines

Here are the steps:

1. Install the NLTK library
2. Run the sentiment analysis

Step 1: Install the NLTK library

NLTK stands for Natural Language ToolKit. It is a library that helps us manage and analyse languages.

We need this as the VADER analyser is part of the NLTK library.

You can install it via Anaconda or Pip.

Next we need to download the VADER Lexicon. Think of this as additional data required to run our VADER analyser.

Run the code below in your Jupyter Notebook to download the vader_lexicon:


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

Step 2: Run the sentiment analysis

It is finally time to run the actual sentiment analysis!

This is the code (it is shorter than you think eh):

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer as SIA

results = []

for headline in df1['Title']:
    pol_score = SIA().polarity_scores(headline) # run analysis
    pol_score['headline'] = headline # add headlines for viewing
    results.append(pol_score)

results
```

We use a loop to pass every headline into our analyser. A sentiment score is assigned to each headline.

Next, we concatenate this list to our original dataframe. However, we are only interested in the values of the 'compound' variable.

```
df1['Score'] = pd.DataFrame(results)['compound']
```

We did it!

4. Correlate lagged score index against prices

In this section, we want to compare the relationship between the TSLA stock returns and our sentiment score. If there is a significant relationship, then our sentiment scores might have some predictive value.

Here are the next steps:

1. Aggregate daily sentiment scores
2. Import TSLA prices and calculate returns
3. Check relationship between lagged score against returns (daily)

Step 1: Aggregate daily sentiment scores

We need only one score per day to compare as TSLA daily prices.

However, there might be more than one article per day. In those cases, we combine the scores for all articles to get a daily score.

```
df2 = df1.groupby(['New Date']).sum() # creates a daily score by summing the
scores of the individual articles in each day
```

The df.groupby() method will remove columns that it deems unnecessary. The output will be the date (as your index) and the daily scores.

Step 2: Import TSLA prices and calculate returns

The goal in this step is to get the daily returns (not stock prices) of TSLA.

Now we need to get the stock prices for TSLA. We will get it from Yahoo Finance manually.

Go to Yahoo finance and search for the TSLA stock ticker.

Click on historical data, choose the dates you want and download the data.

The data will be downloaded as a CSV.

Alternative, if you are lazy, grab it from our repo.

Pandas has a convenient method to import CSV files:

```
import pandas as pd

# Make sure csv file is in the same folder as this notebook
dfEodPrice = pd.read_csv("tsla-eod-prices.csv")
```

Some of you won't know this but the "Date" data is in a string format. You can check with the following code:

```
type(dfEodPrice['Date'][1])
```

Thus, we need to convert the "Date" column to datetime format.

We shall use another method called pd.astype() to do this.

```
dfEodPrice['Date'] = dfEodPrice['Date'].astype('datetime64[ns]')
```

This code will change the entire "Date" column to a datetime format.

Next, since we are only interested in the "Adj Close" column in this article so let's drop all unwanted rows. Next, we set our "Date" column as our index so that it is easier to manage.


```
dfEodPrice2 = dfEodPrice.drop(['Open', 'High','Low','Close','Volume'], axis=1) #
drop unwanted rows
dfEodPrice2.set_index('Date', inplace=True) # set Date coloumn as index
```

Now that we have our prices, we need to calculate our returns.

```
dfEodPrice2['Returns'] = dfEodPrice2['Adj Close']/dfEodPrice2['Adj
Close'].shift(1) - 1 # calculate daily returns
```

To calculate daily returns, we divide today's prices by yesterday's.

Step 3: Check relationship between lagged score against returns (daily)

The goal in this step is to check if the sentiment score predicts future stocks returns.

To do that, we check the relationship between the *one-day lagged* sentiment score and TSLA returns using simple regression.

A one-day lagged sentiment score allows us to compare today's article headlines to tomorrow's stock returns.

This is an important point as we need our score index to predict the future, not to tell us what is happening in the present.

Of course, we can argue that the headline might have an immediate impact on stock prices. To test that, we need accurate price data on a minute or even second timeframe.

We won't do that in this article because it is more difficult to set up that test, minute and second price data is expensive and sometimes inaccurate, there are a lot of variables in live trading (liquidity, spread etc) that may not allow us to enter our trades at the prices stated etc.

Alright, let's start the analysis. Here are the steps:

1. Lagged the sentiment score
2. Match the daily returns with the lagged sentiment score
3. Clean the data (again)
4. Design the test
5. Test for predictive value

Step 1: Lagged the sentiment score

```
df2['Score(1)'] = df2.shift(1)
```

This code shifts all the data down by one row.

Wait are we shifting it down? Shouldn't it be up? It's actually down.

Here is how to think about it. E.g. on 2018-01-16, the lagged score is 0.5719. When we run a regression of 0.5719 against the TSLA's 2018-01-16 returns, we are in fact checking the 2018-01-15's score against 2018-01-16's returns.

This is what we want. Older date's score vs future returns.

Step 2: Match the daily returns with the lagged sentiment score

The number of rows of our score index is not the same as the number of rows of our returns.

shape[0] returns the number of rows

This happens as there are some trading days where there isn't any news.

Thus, we need to match the daily returns against the corresponding sentiment scores before we can run the regression.

```
dfEodPrice3 = pd.merge(dfEodPrice2[['Returns']], df2[['Score(1)']],
left_index=True, right_index=True, how='left')
```

We use the pd.merge() for this purpose. The above code will create a new dateframe that uses TSLA returns as reference and pull the appropriate lagged sentiment score for it.

Think of this as a more complicated version of "vlookup" in Excel, but it does the same thing.

You can learn more about the pd.merge() method here and here.

Step 3: Clean the data (again)

On days where there is no news, there are no sentiment scores. The score column will show a NaN (not-a-number) when there are no scores.

Having a NaN is the equivalent of having a score of 0. Thus, we replace all NaNs with 0.

We do it using this code:

```
dfReturnsScore.fillna(0, inplace=True)
# replace NaN with 0 permanently
```


Step 4: Design the test

The lazy way to run the test is to check the relationship between the daily sentiment scores against TSLA's daily returns.

However, in addition to article headlines, there are many factors affecting TSLA's stock price.

We will not go in-depth on how to isolate the effect of headlines. For now, let's do the bare minimum.

The bare minimum is to exclude the data where the score is 0 or insignificant.

We shall assume that a score of between -0.5 and 0.5 is insignificant for the sake of simplicity. This is an arbitrary figure. You can optimise it in a walk-forward optimisation if you want.

By doing this, we have defined our hypothesis as such:

*A sentiment score of > 0.5 or < -0.5 has a predictive value on **only** tomorrow's TSLA daily returns.*

This test doesn't test if the score has any longer term effects as we are only comparing today's score against tomorrow's stock returns.

The code below removes all data where the sentiment score is between -0.5 and 0.5.

```
dfReturnsScore2 = dfReturnsScore[(dfReturnsScore['Score(1)'] > 0.5) |
(dfReturnsScore['Score(1)'] < -0.5)]
```


Step 5: Test for predictive value

Finally, our data is cleaned and ready for us. Now we need to test if there is a positive relationship between the lagged sentiment score and the daily returns.

There are a few ways to do this:

1. Check for correlation
2. Run a regression
3. Run a cointegration test like the Augmented Dickey–Fuller test
4. Run a hypothesis of means test (you need to log in to Quantopian to see this tutorial)

Any of the above 4 tests will suffice. The reason being, if we are satisfied with the test results, we still need to test the strategy using a production environment with proper backtesting – simulating firing of trades, using in and out-of-sample data, accounting for costs and commission, avoiding overfitting etc.

Thus, you can think of these statistical tests as an early filter to see if we have any potential.

In this article, we shall keep it simple and run a correlation.

Before that, let's plot our data and visualise it.

On the x-axis, we have our 1-day lagged sentiment score. On the y-axis we have our daily TSLA returns.

That doesn't look so good. We want an upward sloping shape. An upward sloping shape indicates that when Score(1) goes up, the daily returns go up, and vice versa.

Ideally, we want something like this:

Anyways, let's run a correlation analysis before we talk about the results. The following code runs a simple correlation calculation.

```
dfReturnsScore2['Returns'].corr(dfReturnsScore2['Score(1)'])
```

Evaluating our results

Our correlation coefficient is 0.044. That's pretty close to 0. This means article headlines alone do not have any predictive value for tomorrow's stock returns.

To be honest, no surprise here. Markets are getting more sophisticated and we ran an overly simplistic analysis.

But no worries, before we end the article, let's look at some improvements we can make to our analysis for real-world trading.

Trading in the Real World – Improving our Analysis

2 second lag and long term relationship

We can split headlines into 2 types. 1) Sensational ones and 2) fundamentals-related ones.

Sensational news

This refers to news that causes an instant impact. If we are doing this, we should use news headlines instead of analysis headlines.

Since the news have an instant impact, if we use a 1-day lag for this, it will be too slow.

Thus, we are better off using a shorter time delay such as a 2 second lag. But note that data of such low timeframes are expensive and might not be accurate.

The bad news is, even if you managed to run this analysis significantly accurately, you will be slaughtered by high frequency, or even regular quantitative hedge funds in the real world as you are competing on speed of execution.

Don't trade on lower timeframes unless you're sure you have an edge.

Fundamentals-related news

This type of news has a longer term fundamental effect. Our SeekingAlpha Analysis headlines fall into this category.

A 1-day lag might be too short for the effect to kick in.

In this case, we can create a long term index score and add or subtract from it based on the individual article headlines. In addition, since newer headlines might have more impact, we can lower the weightage for older headlines.

We can then compare the TSLA prices (not returns) against this index.

Sandbox your strategy

There is a lot of noise in the market. Many factors affect TSLA stock prices in addition to headlines (though the headlines are supposedly an approximate representative of these other factors).

Trading an asset using only headlines when the asset is bombarded by many other factors is dangerous.

As mentioned before in the earlier part of this article, we can alleviate this problem by hedging it with another asset. We then use relative value of sentiment scores as our predictor.

To read more on sandboxing: How to use Hedging as a Trading Strategy

High Impact Dates

Don't trade on days where other variables have huge impact.

If you know that a President election result is being announced today, your SeekingAlpha's Tesla headline is probably not going to have much impact.

If Tesla is announcing their earnings, then non-earnings related articles will not have much impact.

To account for these in your analysis, remove these exogenous high impact dates from your data set.

Accuracy of our sentiment analyser

The accuracy of the VADER sentiment analyser is nowhere near perfect. Just by eyeballing the output, you should be able to see this.

I would say the scores for these 3 headlines are quite off the mark

As mentioned earlier, we already know that these sentiment output have huge variance and we rely on large numbers to squeeze out a slightly useful mean output value.

That said, if you want to improve on this, the solution will be to build your own sentiment analyser by training it on the type of data you are testing on.

Eg. if you are using SeekingAlpha's headlines, train a lexicon-based analyser that is only based on SeekingAlpha's headlines. But be aware that your analyser is overfitted to SeekingAlpha's data and will not work well if applied to something different.

The alternative is to wait 10 years for someone to develop a super accurate sentiment analyser (I'm sure quant funds have already done this) and open source it.

Use delta of the score instead of raw score

Think one step ahead.

Compare the sentiment score with what the current expectations are.

If you know that Tesla is viewed very negatively in the markets, a great score will be more impacted.

If Tesla is already viewed optimistically, then a great score is not as impactful.

Look for headlines from more than one sources

Currently we have only looked at headline data from SeekingAlpha. It might be safer to procure our data from different sources for different purposes.

For sensational news, you would want headlines from the bigger news channels.

For longer term fundamental articles, you might want to procure them from more legitimate blogs or research firms.

In both cases, you will want a mixture from different sources. This will increase objectivity of the data as some sources tend to be biased.

Complement headline data with other data

Headline data is just one aspect.

Combine this data with other alternative data such as satellite/drone images of Tesla's factories, scrape the amount of listings of 2nd hand Tesla cars, activity of their social media etc, to get a better prediction.

Ending Note – Truly understand the trade

At the end of the day, you need to truly understand the reason for your trade.

Come up with a hypothesis and test it appropriately.

Isolate the variables you want to test, split your data into in and out-of-sample pieces, watch out for overfitting or p-hacking.

Here is an interview on the framework to design trading strategies that I find useful.

Trading is a competitive sport. This article covers some basics for sentiment analysis. Think of it as teaching you how each chess piece moves.

To win in trading, you need to learn strategies to outsmart others, since everyone is trying to outwit one another all the time, you need to be creative and keep innovating to stay in the game.

Trading is a hard way to make money. Good luck!

You can download all the code used here: Github repo

Lucas Liew

Founder at AlgoTrading101

Programming Trading