part2-final

January 3, 2019

0.1 # News Analytics and Stock Price Performance: Feature Preprocessing, Exploration and Engineering (part 2)

Can we use news analytics and market data to predict stock price performance? There is no doubt that the ubiquity of data today enables investors at any scale to make better investment decisions but to truly harness this power, we must be able to distinguish signal from noise.

This is a 3 part walkthrough of a Kaggle competition by Two Sigma, with the end result being a model that predicts a signed confidence of an assets fluctuation over a ten-day window.

$$\hat{y}_{ti} \in [-1, 1]$$

Initially, I was a little confused with the evaluation process here. Most people think of stock market predictions as being regression problems but this seemed like a binary classification problem to me. An asset either has a positive or a negative return, with the signed confidence being used to indicate both the direction and the magnitude of this move.

as mentioned above, the signed confidence interval needs to be between [-1 and 1]. Binary classification models are going to output a probability, naturally being a number between 0 and 1. So in order to get the output of my model to conform to this structure, I decided I would multiply my predicted value by 2 and then subtract 1. If the predicted probability is 0, this will cause the output to be -1 and if the predicted probability is 1, then the output will be 1.

For each day in the evaluation time period, we calculate:

$$x_t = \sum_i \hat{y}_{ti} r_{ti} u_{ti}$$

$$score = \frac{\bar{x}_t}{\sigma(x_t)}$$

where r_{ti} is the 10-day market-adjusted leading return for day t for instrument i, and u_{ti} is a 0/1 universe variable that controls whether a particular asset is included in scoring on a particular day.

Your submission score is then calculated as the mean divided by the standard deviation of your daily x_t values:

If the standard deviation of predictions is 0, the score is defined as 0.

Two sources of data for this competition:

Market data (2007 to present) provided by Intrinio - contains financial market information such as opening price, closing price, trading volume, calculated returns, etc.

News data (2007 to present) Source: Thomson Reuters - contains information about news articles/alerts published about assets, such as article details, sentiment, and other commentary.

There are 3 notebooks for this walkthrough. The first includes all of the EDA for both datasets. In the second, I walk through feature preprocessing, exploration and engineering. Finally, in the third notebook, I build, test and tune multiple machine learning models.

```
In [1]: import numpy as np
        import pandas as pd
        import os
        from kaggle.competitions import twosigmanews
        import matplotlib.pyplot as plt
        from matplotlib.pyplot import figure
        import seaborn as sns; sns.set()
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import Imputer
        from sklearn.preprocessing import StandardScaler
        from sklearn import ensemble
        from sklearn.metrics import accuracy_score
        import scipy.stats as stats
        from sklearn.feature_extraction.text import TfidfVectorizer
        import category_encoders as ce
        import warnings
        import gc
        import psutil
        warnings.filterwarnings('ignore')
        %matplotlib inline
```

```
In [2]: def cpuStats():
            pid = os.getpid()
            py = psutil.Process(pid)
            memoryUse = py.memory_info()[0] / 2. ** 30
            return 'memory GB:' + str(np.round(memoryUse, 2))
```

```
In [3]: cpuStats()

Out[3]: 'memory GB:0.25'
```

```
In [4]: # Load training data from API
        env = twosigmanews.make_env()
        (market_train_df, news_train_df) = env.get_training_data()

Loading the data... This could take a minute.
Done!
```

```
In [5]: cpuStats()

Out[5]: 'memory GB:7.04'
```

As you can see, I have no control over how the data is imported and after doing so, I lose half of my available RAM. The two *market_train_df* and *news_train_df* files are stored in the Kaggle environment in feather format.

In [6]: !cat {twosigmanews.__file__}

```
# AUTO-GENERATED FILE, DO NOT MODIFY
"""kaggle.competitions.twosigmanews package

Provides a helper function to create an environment which facilitates
participation in the Two Sigma Financial News Challenge competition.
"""

from kaggle.competitions.twosigmanews import env


def make_env():
    """Returns a new environment supporting the Two Sigma News competition."""
    return env.TwoSigmaNewsEnv()

__all__ = ['make_env']
```

In [7]: market_train_df.info(verbose=True, null_counts=True)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4072956 entries, 0 to 4072955
Data columns (total 16 columns):
time                        4072956 non-null datetime64[ns, UTC]
assetCode                   4072956 non-null object
assetName                   4072956 non-null category
volume                      4072956 non-null float64
close                       4072956 non-null float64
open                        4072956 non-null float64
returnsClosePrevRaw1        4072956 non-null float64
returnsOpenPrevRaw1         4072956 non-null float64
returnsClosePrevMktres1     4056976 non-null float64
returnsOpenPrevMktres1      4056968 non-null float64
returnsClosePrevRaw10       4072956 non-null float64
returnsOpenPrevRaw10        4072956 non-null float64
returnsClosePrevMktres10    3979946 non-null float64
returnsOpenPrevMktres10     3979902 non-null float64
returnsOpenNextMktres10     4072956 non-null float64
universe                    4072956 non-null float64
dtypes: category(1), datetime64[ns, UTC](1), float64(13), object(1)
memory usage: 474.1+ MB
```

In [8]: news_train_df.info(verbose=True, null_counts=True)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9328750 entries, 0 to 9328749
Data columns (total 35 columns):
```

```
time                    9328750 non-null datetime64[ns, UTC]
sourceTimestamp         9328750 non-null datetime64[ns, UTC]
firstCreated            9328750 non-null datetime64[ns, UTC]
sourceId                9328750 non-null object
headline                9328750 non-null object
urgency                 9328750 non-null int8
takeSequence            9328750 non-null int16
provider                9328750 non-null category
subjects                9328750 non-null category
audiences               9328750 non-null category
bodySize                9328750 non-null int32
companyCount            9328750 non-null int8
headlineTag             9328750 non-null object
marketCommentary        9328750 non-null bool
sentenceCount           9328750 non-null int16
wordCount               9328750 non-null int32
assetCodes              9328750 non-null category
assetName               9328750 non-null category
firstMentionSentence    9328750 non-null int16
relevance               9328750 non-null float32
sentimentClass          9328750 non-null int8
sentimentNegative       9328750 non-null float32
sentimentNeutral        9328750 non-null float32
sentimentPositive       9328750 non-null float32
sentimentWordCount      9328750 non-null int32
noveltyCount12H         9328750 non-null int16
noveltyCount24H         9328750 non-null int16
noveltyCount3D          9328750 non-null int16
noveltyCount5D          9328750 non-null int16
noveltyCount7D          9328750 non-null int16
volumeCounts12H         9328750 non-null int16
volumeCounts24H         9328750 non-null int16
volumeCounts3D          9328750 non-null int16
volumeCounts5D          9328750 non-null int16
volumeCounts7D          9328750 non-null int16
dtypes: bool(1), category(5), datetime64[ns, UTC](3), float32(4), int16(13), int32(3), int8(3)
memory usage: 1.1+ GB
```

We can see that these are two fairly large DataFrames. Some of the features are imported with specific types that are memory heavy, such as dates. I do not have control over this.

```
In [9]: print(f'market_train_df: {market_train_df.shape}')
        market_train_df.head()
```

```
market_train_df: (4072956, 16)
```

```
Out[9]:                         time   ...    universe
        0 2007-02-01 22:00:00+00:00   ...         1.0
```

```
1 2007-02-01 22:00:00+00:00   ...         0.0
2 2007-02-01 22:00:00+00:00   ...         1.0
3 2007-02-01 22:00:00+00:00   ...         1.0
4 2007-02-01 22:00:00+00:00   ...         1.0

[5 rows x 16 columns]
```

In [10]:
```
market_train_df.tail()
```

Out[10]:
```
                              time   ...    universe
4072951 2016-12-30 22:00:00+00:00   ...         0.0
4072952 2016-12-30 22:00:00+00:00   ...         0.0
4072953 2016-12-30 22:00:00+00:00   ...         0.0
4072954 2016-12-30 22:00:00+00:00   ...         1.0
4072955 2016-12-30 22:00:00+00:00   ...         1.0

[5 rows x 16 columns]
```

In [11]:
```
print(f'news_train_df: {news_train_df.shape}')
news_train_df.head()
```

```
news_train_df: (9328750, 35)
```

Out[11]:
```
                        time      ...       volumeCounts7D
0 2007-01-01 04:29:32+00:00      ...                    7
1 2007-01-01 07:03:35+00:00      ...                    3
2 2007-01-01 11:29:56+00:00      ...                   17
3 2007-01-01 12:08:37+00:00      ...                   15
4 2007-01-01 12:08:37+00:00      ...                    0

[5 rows x 35 columns]
```

0.2 Chi2 for assetCode

In [12]:
```
column = 'assetCode'
y = market_train_df['returnsOpenNextMktres10'] > 0
print(market_train_df[column].nunique())

cont = pd.crosstab(market_train_df.iloc[:2500][column], y[:2500])
chi2_res = stats.chi2_contingency(cont)
print(chi2_res[1])

cont = pd.crosstab(market_train_df[column], y)
chi2_res = stats.chi2_contingency(cont)
print(chi2_res[1])
```

```
3780
2.5696864141804518e-33
0.0
```

Considering the size of these DataFrames, let's run a chi2 test on the *assetCode* feature to see if it's statistically significant. This may have been a redundant step considering I am dealing with millions of instances and with larger datasets, the P-value will likely start to drift towards a significant value.

Since *assetCode* appears to be statistically significant, I will encode it.

0.3 Helper Functions

Below is a list of helper functions I wrote to keep my code DRY. One thing I have learned from being a software developer is that whenever you find yourself repeating code, create a function. This can go into a toolbox and be used not only across the current application but also across multiple projects.

```
In [13]: def label_cat(df, col):
             return df[col].astype('category').cat.as_ordered()

         def bin_encode(df, cols_to_bin):
             ce_bin = ce.BinaryEncoder(cols = cols_to_bin)
             return ce_bin.fit_transform(df)

         # Find all features with NaN's
         def get_null_features(df):
             return df.columns[df.isna().any()].tolist()

         # Returns either month, day or year of date -> expedites feature engineering
         def get_date_feature(df, col, date_type):
             if (date_type == 'year'):
                 return pd.to_datetime(df[col]).dt.year
             elif (date_type == 'month'):
                 return pd.to_datetime(df[col]).dt.month
             elif (date_type == 'quarter'):
                 return pd.to_datetime(df[col]).dt.quarter
             else:
                 return pd.to_datetime(df[col]).dt.dayofweek

         # Helper function to print accuracy
         def print_accuracy(model, x, y, x_val, y_val):
             print("Training Accuracy Score: ", accuracy_score(model.predict(x), y))
             print("Validation Accuracy Score: ", accuracy_score(model.predict(x_val), y_val))

         # Helper function to get feature importances
         def get_feature_imp(model, x):
             return pd.DataFrame(model.feature_importances_,
                                 index = x.columns,
                                  columns=['importance']).sort_values('importance',

         # Calculate the RSI
         def RSI(series, period):
```

```
    delta = series.diff().dropna()
    u = delta * 0
    d = u.copy()
    u[delta > 0] = delta[delta > 0]
    d[delta < 0] = -delta[delta < 0]
    u[u.index[period-1]] = np.mean( u[:period] ) #first value is sum of avg gains
    u = u.drop(u.index[:(period-1)])
    d[d.index[period-1]] = np.mean( d[:period] ) #first value is sum of avg losses
    d = d.drop(d.index[:(period-1)])
    rs = u.ewm(com=period-1, adjust=False).mean() / d.ewm(com=period-1, adjust=False)
    return 100 - 100 / (1 + rs)

# 80%, 20%
def temploral_split(df, label):
    train, validate = np.split(df, [int(.8*len(df))])

    X_train = train.loc[:, ~train.columns.isin([label])]
    y_train = train[label] > 0

    X_val = validate.loc[:, ~validate.columns.isin([label])]
    y_val = validate[label] > 0

    return [X_train, y_train, X_val, y_val]

# Join 2 dataframes
def join_df(left, right, left_on, right_on=None, suffix='_y'):
    if right_on is None: right_on = left_on
    return left.merge(right, how='left', left_on=left_on, right_on=right_on,
                      suffixes=("", suffix))
```

0.4 Experimenting with Just Market Data

Since stock market predictions have historically and primarily been based on historical market data itself, I wanted to start with just market data and see if the added news data contribute anything significant.

```
In [14]: # Store this for join later on
         tmpAssetCode = market_train_df['assetCode'].values
         tmpMarketDates = market_train_df.time.dt.strftime("%Y%m%d").astype(int)
```

0.5 Handling Categorical Features

```
In [15]: # Handle categorical variables
         market_train_df['assetCode_cat'] = label_cat(market_train_df, 'assetCode').cat.codes

         # I experimented with both label encoding and binary encoding
         # market_train_df = bin_encode(market_train_df, ['assetCode'])
```

I experimented with both label encoding as well as binary encoding. I found that I was getting better results with label encoding. One-hot-encoding is probably not the best approach here considering the high cardinality of the *assetCode* feature, having thousands of unique values.

```
In [16]: market_train_df['assetCode_cat'][0:5]

Out[16]: 0     0
         1     2
         2     7
         3     8
         4    15
         Name: assetCode_cat, dtype: int16

In [17]: # Drop
         market_train_df.drop(['assetCode', 'assetName', 'universe'], axis=1, inplace=True)
```

Since we label encoded *assetCode*, the *assetName* feature is somewhat redundant now.

0.6 Handling Dates

Let's do some feature engineering with the date columns. I'll start with the day of week and month.

```
In [18]: # Create year, month, day features
         market_train_df['month'] = get_date_feature(market_train_df, 'time', 'month')
         market_train_df['day'] = get_date_feature(market_train_df, 'time', 'day')
         market_train_df['quarter'] = get_date_feature(market_train_df, 'time', 'quarter')

         # Time to drop the time feature
         market_train_df.drop(['time'], axis=1, inplace=True)

In [19]: market_train_df.head()

Out[19]:        volume  close   open   ...     month  day  quarter
         0   2606900.0  32.19  32.17   ...         2    3        1
         1   2051600.0  11.12  11.08   ...         2    3        1
         2   1164800.0  37.51  37.99   ...         2    3        1
         3  23747329.0  84.74  86.23   ...         2    3        1
         4   1208600.0  18.02  18.01   ...         2    3        1

         [5 rows x 16 columns]
```

0.7 Handling Numerical features

```
In [20]: market_train_df.isnull().sum()

Out[20]: volume                          0
         close                           0
         open                            0
```

```
returnsClosePrevRaw1            0
returnsOpenPrevRaw1             0
returnsClosePrevMktres1     15980
returnsOpenPrevMktres1      15988
returnsClosePrevRaw10           0
returnsOpenPrevRaw10            0
returnsClosePrevMktres10    93010
returnsOpenPrevMktres10     93054
returnsOpenNextMktres10         0
assetCode_cat                   0
month                           0
day                             0
quarter                         0
dtype: int64
```

In [21]:
```
mrkt_null = get_null_features(market_train_df)
mrkt_null
```

Out[21]:
```
['returnsClosePrevMktres1',
 'returnsOpenPrevMktres1',
 'returnsClosePrevMktres10',
 'returnsOpenPrevMktres10']
```

In [22]:
```
market_train_df[mrkt_null].describe()
```

Out[22]:
```
       returnsClosePrevMktres1           ...              returnsOpenPrevMktres10
count             4.056976e+06           ...                         3.979902e+06
mean              1.738580e-04           ...                         1.481702e-02
std               3.270305e-02           ...                         7.285742e+00
min              -1.235622e+00           ...                        -1.375045e+03
25%              -8.569246e-03           ...                        -2.962645e-02
50%              -1.236127e-04           ...                         1.126206e-03
75%               8.397528e-03           ...                         3.171535e-02
max               4.512244e+01           ...                         9.761338e+03

[8 rows x 4 columns]
```

We can see from the min / max / std that these columns need to be standardized.

0.8 Impute missing values

In [23]:
```
imp = Imputer(missing_values='NaN', strategy='median', axis=0)
imp = imp.fit(market_train_df[mrkt_null])
market_train_df[mrkt_null] = imp.transform(market_train_df[mrkt_null])
```

We have 4 columns with NaN's and the number of NaN's is not substantial so let's impute the missing values with the medians.

In [24]:
```
# No more NaN's
get_null_features(market_train_df)
```

Out[24]:
```
[]
```

0.9 Feature Engineering

```
In [25]: for n in [14, 30, 50, 200]:

             # Create the moving averages
             market_train_df['close_ma' + str(n)] = market_train_df['close'].rolling(window=n)

             # Create RSI
             market_train_df['rsi' + str(n)] = RSI(market_train_df['close'], n)
```

Here I am creating moving averages and RSI's for 14, 30, 50 and 200-day windows. Moving average is self-explanatory. RSI stands for relative strength index and can be interpreted as follows:

When the RSI is close to 0, it might indicate that the price of an asset is due to rebound because of recent lows. When the RSI is close to 100, it might indicate that the price of an asset is due to decrease because of recent highs.

RSI = 100 - (100 / 1 + RS)

RS = avg gain over n periods / avg loss over n periods

```
In [26]: market_train_df.head()

Out[26]:        volume  close   open  ...   rsi50  close_ma200  rsi200
         0   2606900.0  32.19  32.17  ...     NaN          NaN     NaN
         1   2051600.0  11.12  11.08  ...     NaN          NaN     NaN
         2   1164800.0  37.51  37.99  ...     NaN          NaN     NaN
         3  23747329.0  84.74  86.23  ...     NaN          NaN     NaN
         4   1208600.0  18.02  18.01  ...     NaN          NaN     NaN

         [5 rows x 24 columns]
```

```
In [27]: print(f'range min: {market_train_df[market_train_df.isnull().any(axis=1)].index.min()
         print(f'range max: {market_train_df[market_train_df.isnull().any(axis=1)].index.max()

range min: 0
range max: 199
```

```
In [28]: # This will be the first 200 rows because of the moving average and RSI calculations
         market_train_df.dropna(inplace=True)

         # Drop from placeholder features for join later
         tmpAssetCode = tmpAssetCode[200:]
         tmpMarketDates = tmpMarketDates[200:]

         market_train_df.head()

Out[28]:        volume  close   open  ...      rsi50  close_ma200     rsi200
         200  858917.0  37.64  37.85  ...  49.808506     41.46755  50.050876
         201  221700.0  52.77  52.39  ...  50.402479     41.67580  50.192277
```

```
202   604100.0   55.41   55.71    ...      50.506764     41.76530  50.216992
203   272900.0  172.27  168.80    ...      54.799557     42.20295  51.292196
204  2451400.0   27.19   27.40    ...      49.374409     42.24880  49.946239

[5 rows x 24 columns]
```

Since the largest window for the moving averages and RSI's is 200, it means we will end up with 200 NaN rows at the beginning of out DataFrame. Let's drop them.

0.10 Standardization

```
In [29]: cols_to_strd = [
             'volume',
             'close',
             'open',
             'returnsClosePrevRaw1',
             'returnsOpenPrevRaw1',
             'returnsClosePrevMktres1',
             'returnsOpenPrevMktres1',
             'returnsClosePrevRaw10',
             'returnsOpenPrevRaw10',
             'returnsClosePrevMktres10',
             'returnsOpenPrevMktres10',
             'close_ma14',
             'rsi14',
             'close_ma30',
             'rsi30',
             'close_ma50',
             'rsi50',
             'close_ma200',
             'rsi200',
         ]
```

```
In [30]: scaler = StandardScaler()
         scaler = scaler.fit(market_train_df[cols_to_strd])
         market_train_df[cols_to_strd] = scaler.transform(market_train_df[cols_to_strd])
```

The ranges in stock market data can be quite extreme. Just think about the volume feature. Let's standardize them.

```
In [31]: market_train_df.head()
```

```
Out[31]:         volume     close      open    ...       rsi50  close_ma200    rsi200
         200 -0.234972 -0.049004 -0.043703    ...   -0.165994     0.210697  0.168992
         201 -0.317860  0.308773  0.297517    ...    0.352261     0.235693  0.637992
         202 -0.268118  0.371201  0.375430    ...    0.443252     0.246436  0.719964
         203 -0.311200  3.134576  3.029388    ...    4.188816     0.298968  4.286191
         204 -0.027826 -0.296114 -0.288940    ...   -0.544754     0.304471 -0.178066

         [5 rows x 24 columns]
```

```
In [32]: market_train_df['vol_pct_change'] = market_train_df['volume'].pct_change()
         market_train_df['vol_pct_10'] = market_train_df['vol_pct_change'].rolling(window=10).r
         market_train_df.drop(['vol_pct_change'], axis=1, inplace=True)
```

During EDA, we saw that volume is essentially uncorrelated to price. Let's try experimenting with volumes and percent changes and see if that uncovers anything.

```
In [33]: print(f'range min: {market_train_df[market_train_df.isnull().any(axis=1)].index.min()
         print(f'range max: {market_train_df[market_train_df.isnull().any(axis=1)].index.max()

range min: 200
range max: 209
```

```
In [34]: market_train_df.dropna(inplace=True)

         # Again, drop the first 10 because of the 10 day window
         # These are added back to market data later on for the DF join
         tmpAssetCode = tmpAssetCode[10:]
         tmpMarketDates = tmpMarketDates[10:]
```

1 Training

```
In [35]: label_counts = market_train_df['returnsOpenNextMktres10'] > 0
         label_counts.value_counts()

Out[35]: True     2078410
         False    1994336
         Name: returnsOpenNextMktres10, dtype: int64
```

We are dealing with a balanced dataset, so accuracy can be used as an effective evaluation metric.

```
In [36]: # Get temporal training splits
         X_train_sample, y_train_sample, X_val_sample, y_val_sample = temploral_split(market_t

In [37]: print(f'X_train_sample: {X_train_sample.shape}')
         print(f'X_val_sample: {X_val_sample.shape}')

X_train_sample: (160000, 24)
X_val_sample: (40000, 24)
```

Since we are dealing with large DataFrames, I am going to start training my models on a small sample of the data. I think it's important to be able to iterate quickly in the early phases and this will enable me to experiment quicker. I'll start by taking the last (most recent) 200,000 records with an 80/20 split.

1.1 Random Forest

I typically start with a Random Forest model for most machine learning problems I encounter for 2 reasons:

1. Ease of use/interpretability
2. Makes almost no assumptions about the data.

```
In [38]: rfc = ensemble.RandomForestClassifier(n_estimators=30, min_samples_leaf=10, max_featu
         %time rfc.fit(X_train_sample, y_train_sample)
```

```
CPU times: user 1min 50s, sys: 100 ms, total: 1min 50s
Wall time: 29.2 s
```

```
Out[38]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features=0.5, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=10, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=-1,
                     oob_score=False, random_state=23, verbose=0, warm_start=False)
```

```
In [39]: print_accuracy(rfc, X_train_sample, y_train_sample, X_val_sample, y_val_sample)
```

```
Training Accuracy Score:  0.9449625
Validation Accuracy Score:  0.53705
```

Not bad for the initial base result. We know there is predictive power in the data. It is overfitting quite badly though. Let's try adjusting the *min_samples_leaf* hyperparameter.

```
In [40]: rfc = ensemble.RandomForestClassifier(n_estimators=30, min_samples_leaf=100, max_featu
         %time rfc.fit(X_train_sample, y_train_sample)
```

```
CPU times: user 1min 26s, sys: 16 ms, total: 1min 26s
Wall time: 22.5 s
```

```
Out[40]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features=0.5, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=100, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=-1,
                     oob_score=False, random_state=23, verbose=0, warm_start=False)
```

```
In [41]: print_accuracy(rfc, X_train_sample, y_train_sample, X_val_sample, y_val_sample)
```

```
Training Accuracy Score:  0.6986
Validation Accuracy Score:  0.535975
```

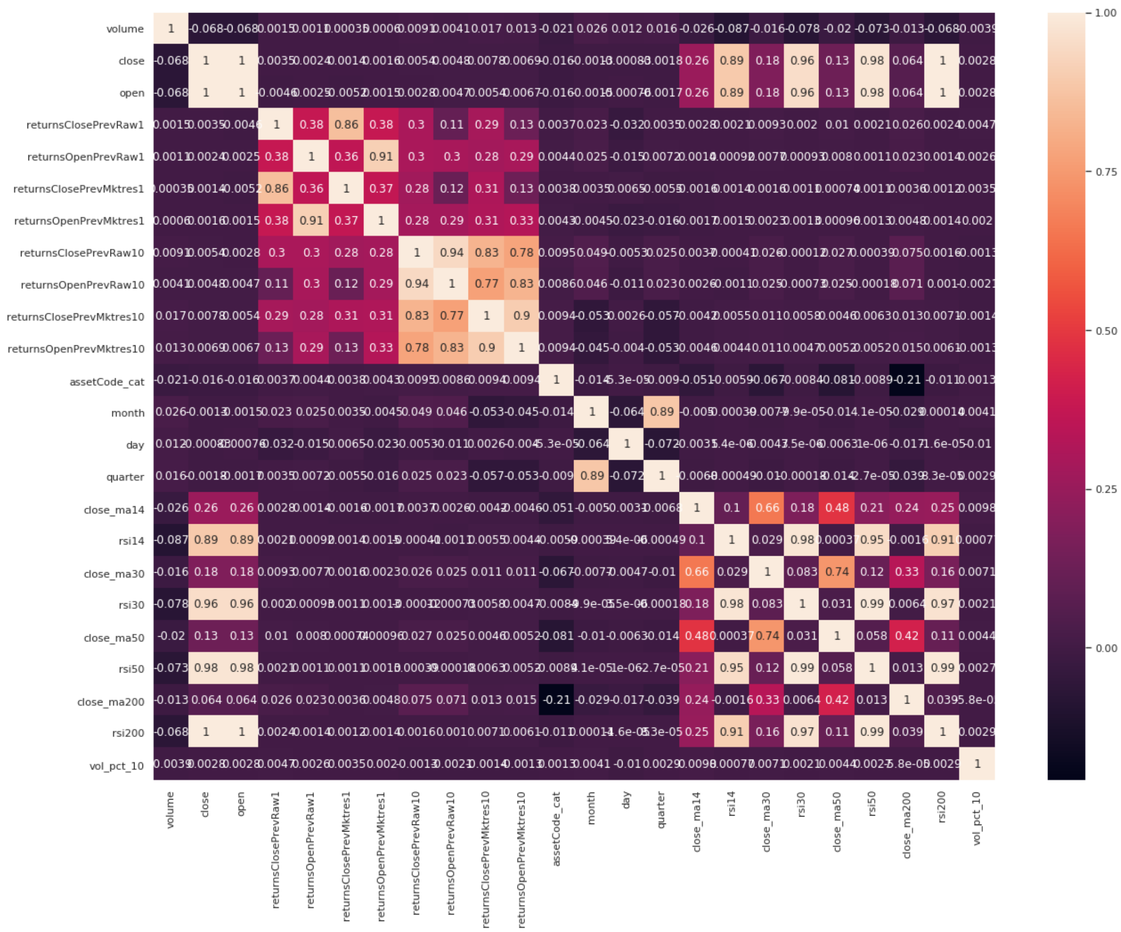
In [42]:
```
feature_importances = get_feature_imp(rfc, X_train_sample)
feature_importances[0:20]
```

Out[42]:

	importance
returnsOpenPrevMktres10	0.087694
returnsOpenPrevRaw10	0.082835
month	0.065618
returnsClosePrevRaw10	0.058592
returnsClosePrevMktres10	0.056799
assetCode_cat	0.056602
volume	0.050824
close_ma200	0.048702
close_ma14	0.047997
close_ma50	0.043122
close_ma30	0.042418
rsi14	0.039586
rsi30	0.034670
returnsOpenPrevMktres1	0.033706
rsi200	0.032567
close	0.030562
rsi50	0.029960
returnsClosePrevMktres1	0.029790
open	0.028920
returnsOpenPrevRaw1	0.028608

It seems suspicious that the month feature is so important. Let's inspect.

In [43]:
```
corr_matrix = X_train_sample.corr()
sns.set(rc={'figure.figsize':(20, 15)})
_ = sns.heatmap(corr_matrix, annot=True)
```



There seem to be a lot of highly correlated features. Let's drop features that are more than 90% correlated.

```
In [44]: # Correlated features to be dropped
         upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))
         to_drop = [column for column in upper.columns if any(upper[column] > 0.90)]
         print(f'Number of correlated features to drop: {len(to_drop)}')
```

```
Number of correlated features to drop: 7
```

```
In [45]: to_drop
```

```
Out[45]: ['open',
          'returnsOpenPrevMktres1',
          'returnsOpenPrevRaw10',
          'returnsOpenPrevMktres10',
          'rsi30',
          'rsi50',
          'rsi200']
```

```
In [46]: X_train_sample.drop(to_drop, axis=1, inplace=True)
         X_val_sample.drop(to_drop, axis=1, inplace=True)
```

```
In [47]: corr_matrix = X_train_sample.corr()
         sns.set(rc={'figure.figsize':(16.7,8.27)})
         _ = sns.heatmap(corr_matrix, annot=True)
```

This looks much better.

```
In [48]: rfc = ensemble.RandomForestClassifier(n_estimators=30, min_samples_leaf=100, max_featu
         %time rfc.fit(X_train_sample, y_train_sample)

CPU times: user 52.6 s, sys: 20 ms, total: 52.6 s
Wall time: 13.8 s


Out[48]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features=0.5, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=100, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=-1,
                     oob_score=False, random_state=23, verbose=0, warm_start=False)
```

```
In [49]: print_accuracy(rfc, X_train_sample, y_train_sample, X_val_sample, y_val_sample)

Training Accuracy Score:  0.69553125
Validation Accuracy Score:  0.531575
```

After dropping 7 features, our accuracy hasn't really been affected.

```
In [50]: feature_importances = get_feature_imp(rfc, X_train_sample)
         feature_importances
```

```
Out[50]:                           importance
         returnsClosePrevRaw10       0.119597
         returnsClosePrevMktres10    0.114182
         close                       0.076727
         rsi14                       0.073290
         assetCode_cat               0.072896
         month                       0.064383
         close_ma14                  0.063179
         close_ma200                 0.061431
         close_ma30                  0.060869
         volume                      0.057028
         close_ma50                  0.055539
         returnsClosePrevMktres1     0.047485
         returnsClosePrevRaw1        0.036277
         returnsOpenPrevRaw1         0.036137
         vol_pct_10                  0.028148
         day                         0.020215
         quarter                     0.012616
```

This looks more like what we would expect.

```
In [51]: # Clear up some memory
         del X_train_sample
         del y_train_sample
         del X_val_sample
         del y_val_sample
```

1.2 ## Time for Some News Data

Now, this DataFrame is a beast. With over 9 million rows, I am coming close to running out of memory.

```
In [52]: print(f'market_train_df: {market_train_df.shape}')
         print(f'tmpAssetCode: {tmpAssetCode.shape}')
         print(f'tmpMarketDates: {tmpMarketDates.shape}')
```

```
market_train_df: (4072746, 25)
tmpAssetCode: (4072746,)
tmpMarketDates: (4072746,)
```

```
In [53]: # Add back the assetCode and time feature so we can join the 2 DataFrames
         market_train_df['assetCode'] = tmpAssetCode
         market_train_df['time'] = tmpMarketDates
```

```
    # Save memory
    del tmpAssetCode
    del tmpMarketDates
```

In [54]:
```
market_train_df.head()
```

Out[54]:
```
       volume     close      open    ...     vol_pct_10  assetCode      time
210 -0.320595 -0.624333 -0.619364    ...      -1.287295     BRKL.O  20070201
211 -0.252896 -0.608016 -0.603406    ...      -1.343687     BRKS.O  20070201
212 -0.247280  0.342588  0.328260    ...      -1.330259      BRL.N  20070201
213 -0.260990 -0.261353 -0.266880    ...      -1.340783      BRO.N  20070201
214 -0.331453 -0.044984 -0.051916    ...      -1.222726      BRS.N  20070201

[5 rows x 27 columns]
```

In [55]:
```
news_train_df['time'] = news_train_df.time.dt.strftime("%Y%m%d").astype(int)
news_train_df['assetCode'] = news_train_df['assetCodes'].map(lambda x: list(eval(x))[(

# determine the proportion of the news item discussing the asset
news_train_df['coverage'] = news_train_df['sentimentWordCount'] / news_train_df['word(

# relative position of the first mention in the item
news_train_df['position'] = news_train_df['firstMentionSentence'] / news_train_df['ser
```

Here I am creating a few NLP features.

sentimentWordCount is the number of lexical tokens in the sections of the item text that are deemed relevant to the asset. This can be used in conjunction with *wordCount* to determine the proportion of the news item discussing the asset.

sentenceCount and *firstMentionSentence* are used to determine to determine the relative position of the first mention in the item.

In [56]:
```
news_train_df.head()
```

Out[56]:
```
       time           sourceTimestamp    ...     coverage  position
0  20070101 2007-01-01 04:29:32+00:00    ...     0.265455  0.545455
1  20070101 2007-01-01 07:03:34+00:00    ...     0.068357  0.145455
2  20070101 2007-01-01 11:29:56+00:00    ...     0.172680  0.933333
3  20070101 2007-01-01 12:08:37+00:00    ...     0.255385  0.928571
4  20070101 2007-01-01 12:08:37+00:00    ...     0.313846  0.785714

[5 rows x 38 columns]
```

In [57]:
```
# get rid of some features from news data
news_drops = ['sourceTimestamp','firstCreated','sourceId','takeSequence','firstMentio
            'headlineTag','subjects','audiences',
            'assetName', 'assetCodes']

news_train_df.drop(news_drops, axis=1, inplace=True)
```

Here we are dropping a few seemingly unnecessary features. *sourceTimestamp* and *firstCreated* are essentially the same things and *sourceId* is a unique value for each row.

1.3 TF-IDF

One of the features on the news DataFrame is the "headline" feature. I thought I would experiment with tf-idf vectors here.

```
In [58]: v = TfidfVectorizer(stop_words='english')
         tfidf = v.fit_transform(news_train_df['headline'])

         # Average across to get a scalar from the vector
         tfidf_mean = tfidf.mean(axis=1)
         del tfidf
```

```
In [59]: news_train_df['tfidf_mean'] = tfidf_mean
         news_train_df.drop(['headline'], axis=1, inplace=True)
         del tfidf_mean
```

```
In [60]: news_train_df.head()
```

```
Out[60]:        time  urgency provider     ...      coverage  position  tfidf_mean
         0  20070101        3     RTRS     ...      0.265455  0.545455    0.000010
         1  20070101        3     RTRS     ...      0.068357  0.145455    0.000008
         2  20070101        3     RTRS     ...      0.172680  0.933333    0.000009
         3  20070101        3     RTRS     ...      0.255385  0.928571    0.000009
         4  20070101        3     RTRS     ...      0.313846  0.785714    0.000009

         [5 rows x 28 columns]
```

```
In [61]: # aggregate -> combine news reports for same assets on same day
         newsgp = news_train_df.groupby(['time','assetCode'], sort=False).aggregate(np.mean).re
         del news_train_df
```

```
In [62]: ts_df = market_train_df.merge(newsgp, how='left', left_on=['time', 'assetCode'], righ
                                       suffixes=("", '_y'))

         del market_train_df
```

Above we simply grouped the news data by *time* and *assetCode*, ran aggregate to get the averages of the data for a given day and then joined to the market data on *time* and *assetCode* features.

```
In [63]: print(f'ts_df: {ts_df.shape}')
         ts_df.head()
```

```
ts_df: (4072746, 52)
```

```
Out[63]:      volume     close      open     ...      coverage  position  tfidf_mean
         0 -0.320595 -0.624333 -0.619364     ...           NaN       NaN         NaN
         1 -0.252896 -0.608016 -0.603406     ...           NaN       NaN         NaN
         2 -0.247280  0.342588  0.328260     ...           NaN       NaN         NaN
         3 -0.260990 -0.261353 -0.266880     ...           NaN       NaN         NaN
         4 -0.331453 -0.044984 -0.051916     ...           NaN       NaN         NaN

         [5 rows x 52 columns]
```

In [64]:
```
ts_df.drop(['assetCode', 'time'], axis=1, inplace=True)
```

In [65]:
```
ts_df[newsgp.columns[2:].values] = ts_df[newsgp.columns[2:].values].fillna(value=0)
del newsgp
```

There will be many assets that will have many days without news data. Let's fill these NaN's with a 0.

1.4 Model Time

In [66]:
```
X_train_sample, y_train_sample, X_val_sample, y_val_sample = temploral_split(ts_df.il
```

In [67]:
```
print(f'X_train_sample: {X_train_sample.shape}')
print(f'X_val_sample: {X_val_sample.shape}')
```

```
X_train_sample: (160000, 49)
X_val_sample: (40000, 49)
```

In [68]:
```
rfc = ensemble.RandomForestClassifier(n_estimators=30, min_samples_leaf=100, max_featu
%time rfc.fit(X_train_sample, y_train_sample)
```

```
CPU times: user 1min 37s, sys: 48 ms, total: 1min 37s
Wall time: 25.8 s
```

Out[68]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features=0.5, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=100, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=-1,
            oob_score=False, random_state=23, verbose=0, warm_start=False)
```

In [69]:
```
print_accuracy(rfc, X_train_sample, y_train_sample, X_val_sample, y_val_sample)
```

```
Training Accuracy Score:  0.6997625
Validation Accuracy Score:  0.53645
```

In [70]:
```
feature_importances = get_feature_imp(rfc, X_train_sample)
feature_importances[0:30]
```

Out[70]:

	importance
returnsOpenPrevMktres10	0.082746
returnsOpenPrevRaw10	0.079236
month	0.061304
returnsClosePrevRaw10	0.060350
returnsClosePrevMktres10	0.058017
assetCode_cat	0.051494
close_ma200	0.048169

```
volume                        0.046338
close_ma14                    0.045030
close_ma30                    0.042122
close_ma50                    0.041928
rsi14                         0.038789
returnsOpenPrevMktres1        0.035483
rsi30                         0.032987
close                         0.032399
rsi200                        0.030941
rsi50                         0.030786
returnsClosePrevMktres1       0.029592
open                          0.028724
returnsOpenPrevRaw1           0.027211
returnsClosePrevRaw1          0.024865
vol_pct_10                    0.020305
day                           0.019475
quarter                       0.012626
volumeCounts7D                0.002642
sentimentNeutral              0.001907
volumeCounts5D                0.001593
sentimentNegative             0.001389
position                      0.001257
coverage                      0.001005
```

The news data doesn't seem to have added much to our model. Perhaps I will have better luck with other models. In the next notebook I will explore tuning my models further.