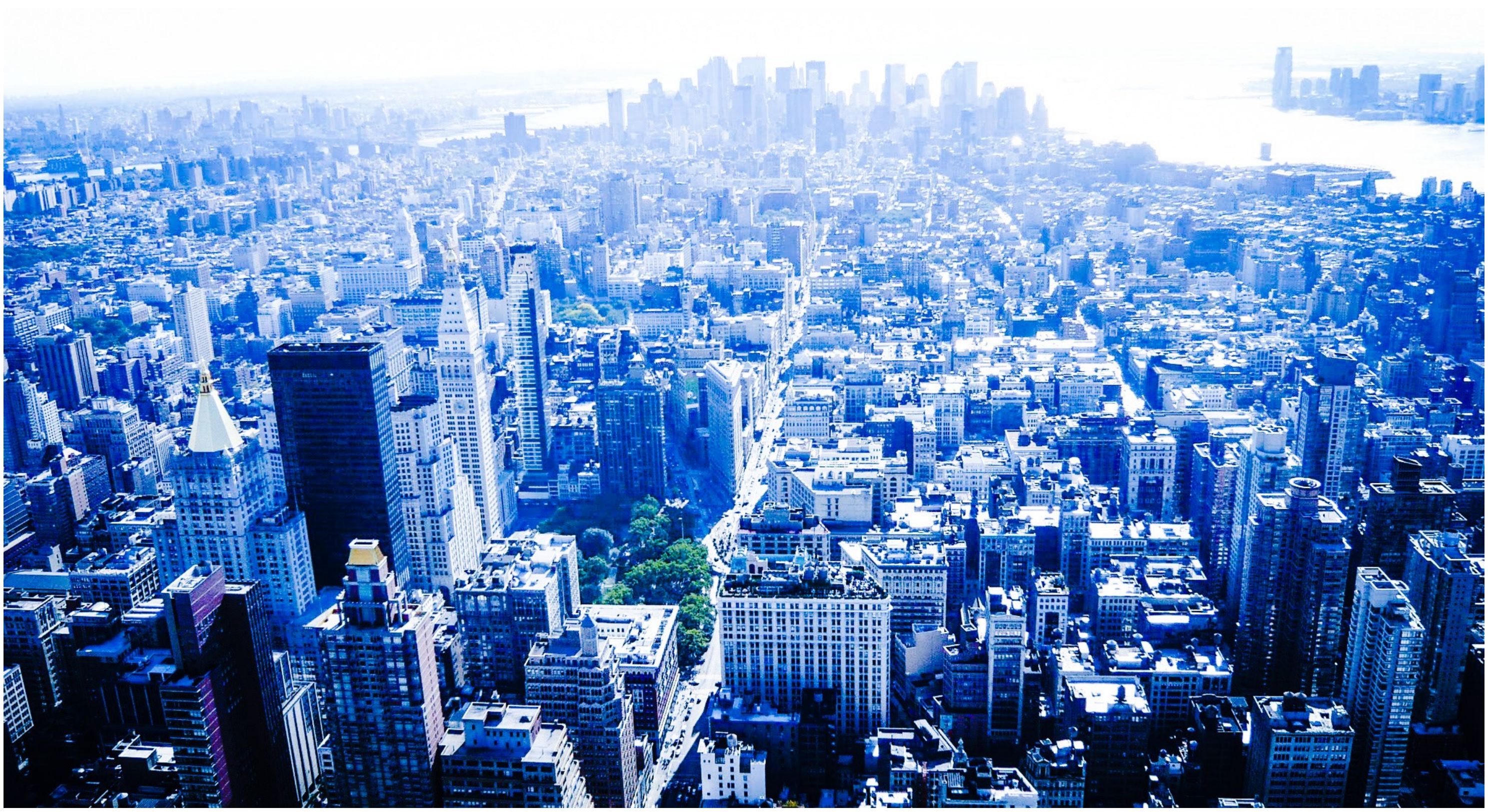
HANDS-ON TUTORIALS

FedSpeak — How to build a NLP pipeline to predict central bank policy changes

A guide to analyse policymakers’ conversations by deep neural network

Yuki Takahashi Nov 13, 2020 · 12 min read ★



(Image Created by Author)

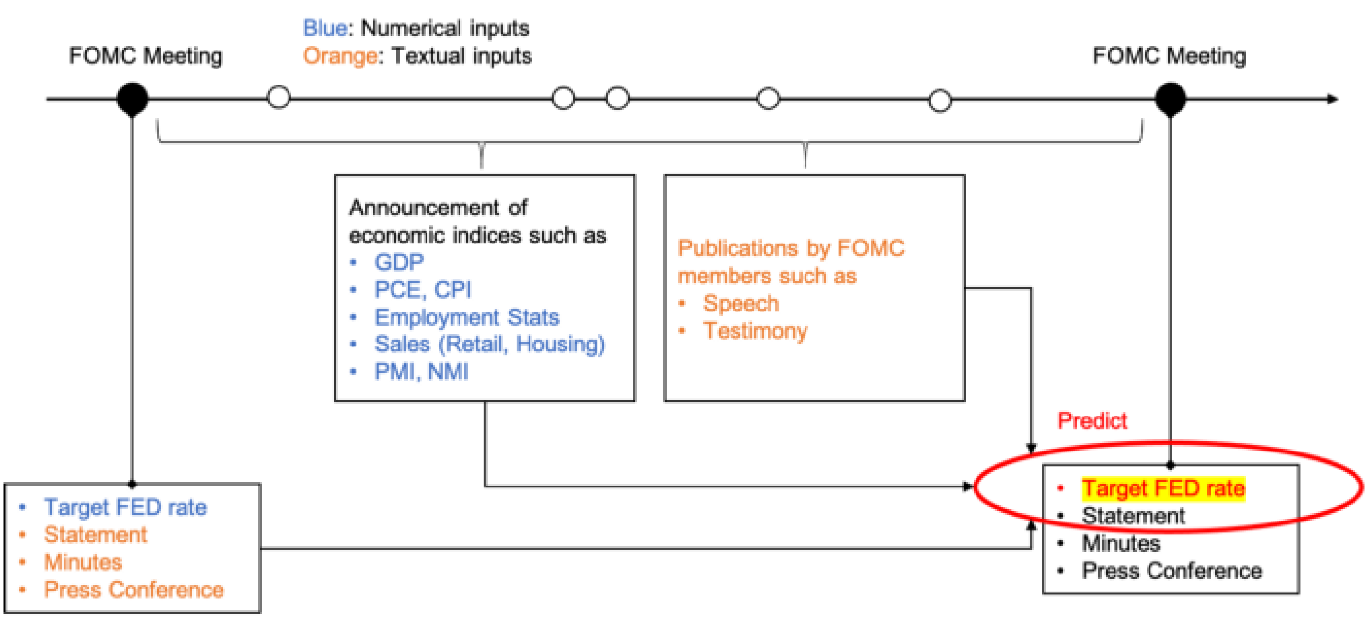
Introduction

This blog describes how I analysed central bank policy by means of NLP techniques in a past project. The source code is available in github repo.

Business Context

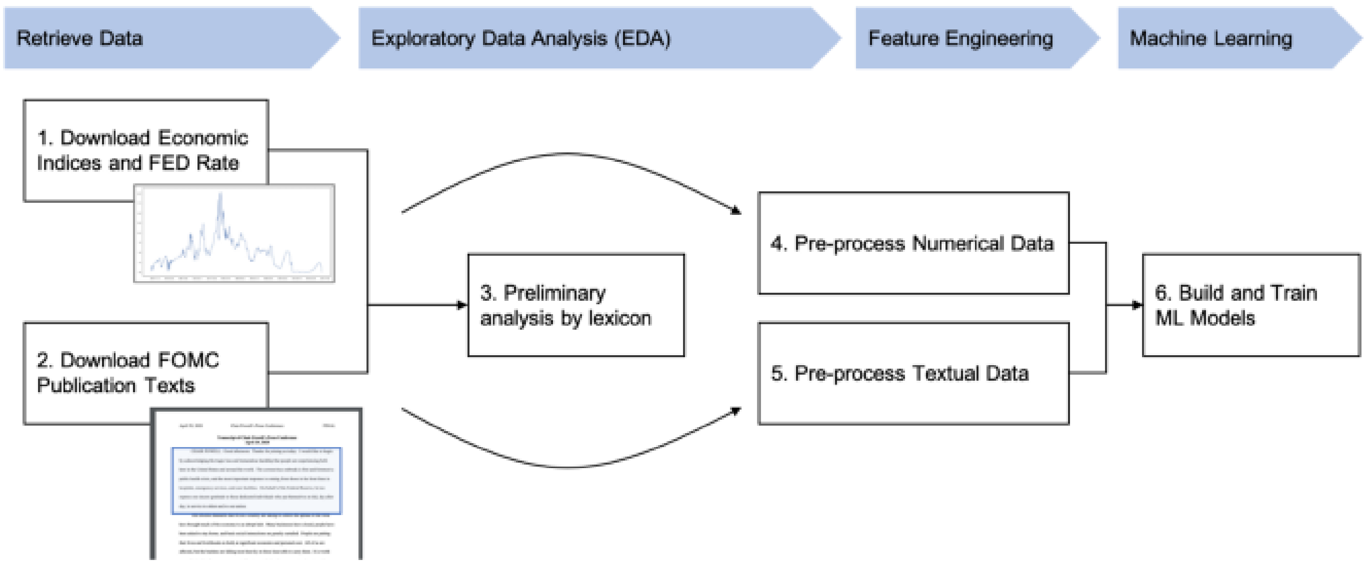
FOMC has eight regular meetings to determine the monetary policy. At each meeting, it publishes press conference minutes, statements as well as scripts in the website. In addition to this regular meetings, the members' speeches and testimonies are also scripted on the website.

At a meeting, the policy makers discuss, vote and decide the monetary policy and publish the decision along with their view on current economic situation and forecast, including Forward Guidance since 2012. The central banks intend to indicate their potential future monetary policy in their publications as a measure of market communication.



The prediction inputs (Created by Author)

The objective of this project is to find latent features in those texts published by FOMC. First, I applied machine learning to economic indices to see the performance of prediction on those numerical data. Then, added pre-processed text data as additional feature in traditional machine learning technique to see if it contains the meaningful information. Finally, apply Deep Learning technique such as LSTM/RNN and BERT to see if these can better predict the rate hike/lower at each FOMC meeting.



Overall process of this project (Created by Author)

1. Retrieving Market Data

Daily FED Rate and major economic indices can be obtained from Economic Research in FRB of St. Louis website called FRED:

- FED Rate
- GDP
- CPI / PCE
- Employment and Unemployment
- Retail Sales and Home Sales

Manufacturing PMI and Service PMI (formerly known as "Non-Manufacturing Index or NMI) are published by ISM (Institute for Supply Management) website.

Daily Treasury yield rates can be downloaded from US Treasury website in xml.

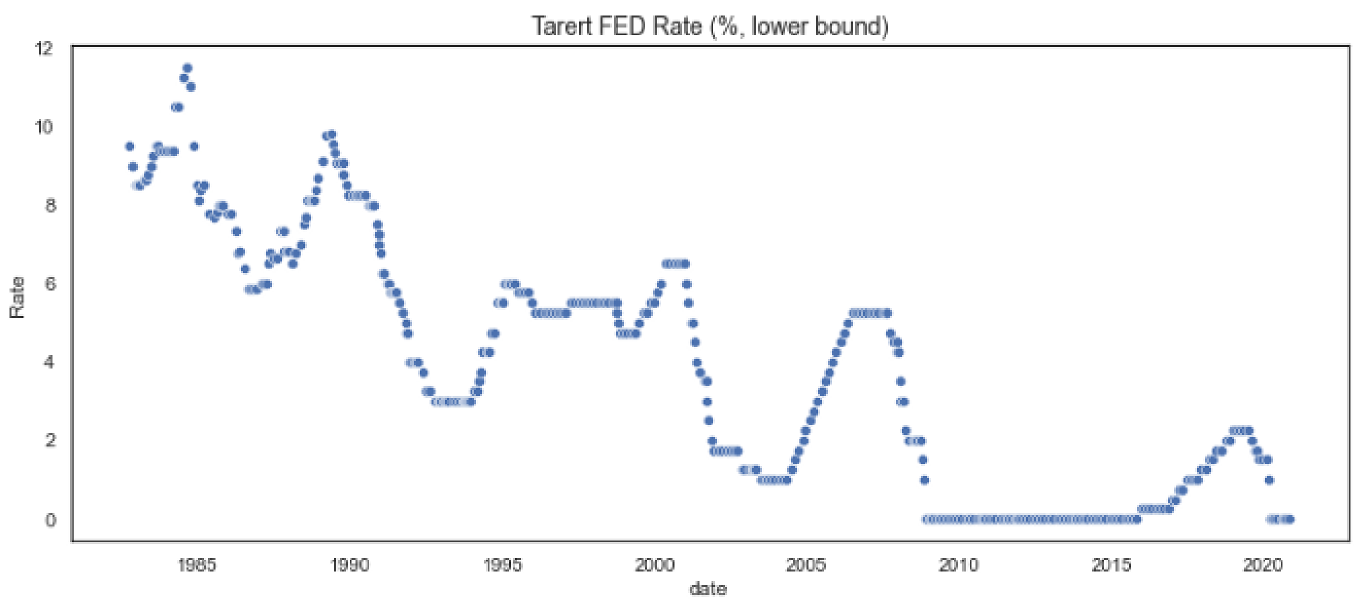
Good to explore the details of the data on each website but it's much more convenient to use Quandl, which provides Web APIs and Libraries to retrieve all the data in the same manner. All the data above are publicly available and free for personal use but you should always check the license terms in the original source in accordance to your objective.

Once you create an Quandl Account, API Key is provided. For example, you can download data in python after `pip install quandl` like this:

```
import sys
import quandl

if __name__ == '__main__':
  quandl.ApiConfig.api_key = sys.argv[1]
  quandl_code = 'FRED/DFEDTARL'
  from_date = '1982-01-01'

  data = quandl.get(quandl_code, start_date=from_date)
  data.to_csv('download.csv')
```

get_quandl_data.py hosted with ♡ by GitHub view raw

FRB changed the target FED Rate to a range instead of a single rate in 2008, so concatenate two series proves either lower bound or upper bound view.



FED Fund Rate, Target Lower Bound (Created by Author)

2. Retrieving Text Data

All FOMC publications are available in FOMC Website. You may notice the website contains materials for each meeting but the contents change over the time. Also, there are unscheduled meetings and conference calls in addition to regular meeting. Some texts are in html while others are only in pdf files. There's even a different website for historical data and the page structure varies.

Download the following texts from FOMC Meeting Calendar:

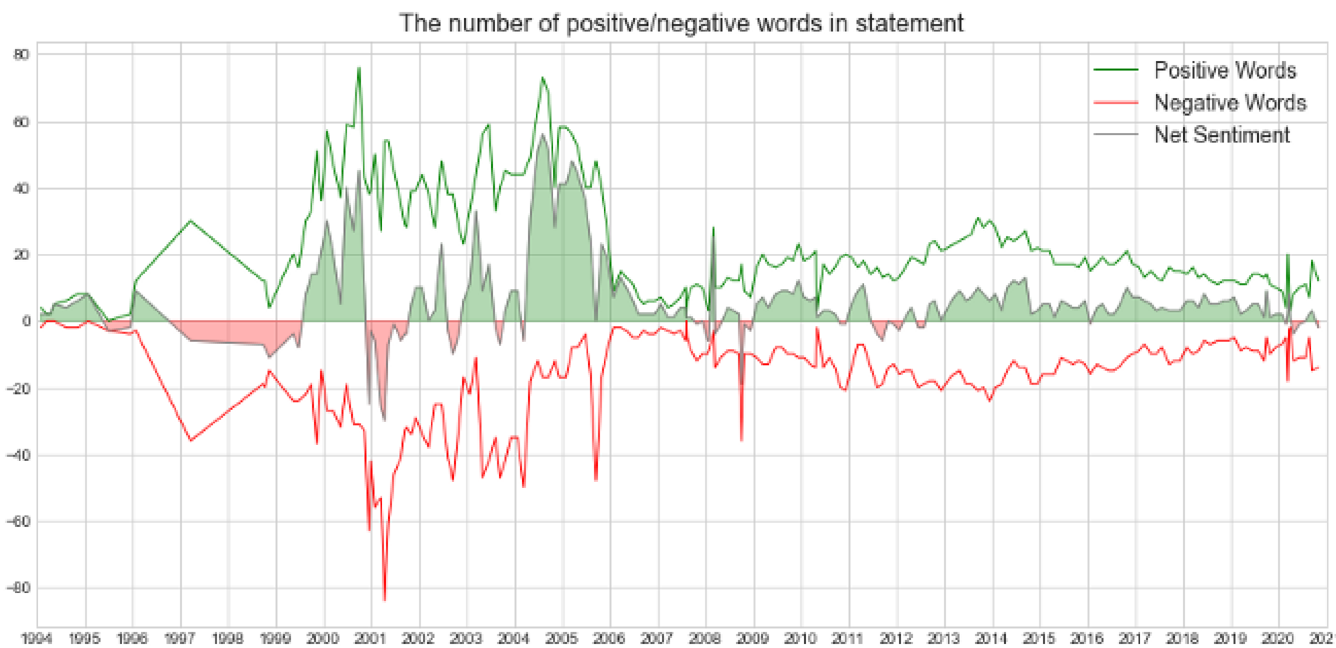
- **Statements** — available right after each FOMC meeting
- **Meeting Minutes** — available three weeks after each FOMC meeting, so may not be available for the latest Meeting
- **Press Conference Transcripts** — available at each FOMC meeting but only started in 2011
- **Meeting Transcripts** — available five years after the meeting, so this cannot be used as input for the prediction while still good source to see the detail background for the old meetings
- **Speeches** — transcripts are published in this page and I used chair's speech published between two meetings
- **Testimony** — a various testimony texts are also published in this page and I used Semiannual Monetary Policy Report to the Congress

When text is in HTML, BeautifulSoup will do the job. Use textract to extract PDF and re module for searching by regular expression.

3. Preliminary Analysis

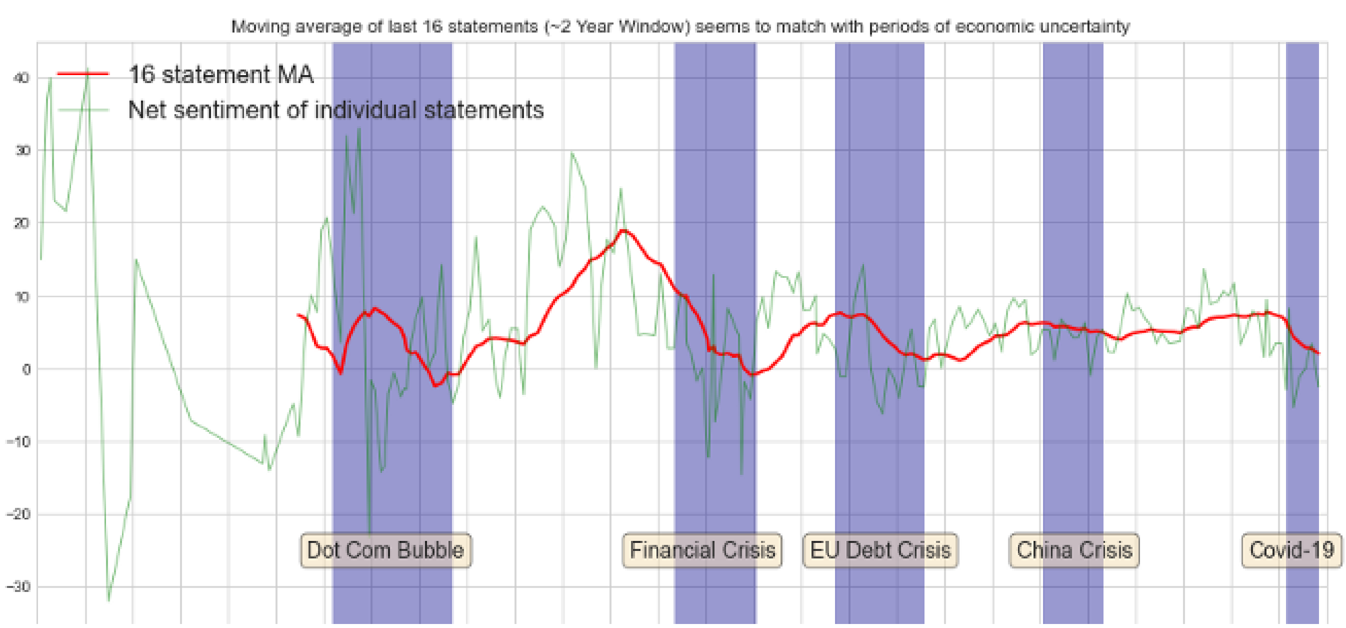
In order to see if the texts may contain some useful insight to predict FED rate, I used Loughran and McDonald Sentiment Word List to measure the sentiment of statement. This dictionary contains several thousands words appearing in financial documents such as 10K, 10Q and earnings calls categorised to positive, negative, etc. It includes words in different forms, so stemming or lemmatising should not be applied. I applied a simply technique to flip the sentiment for negation (e.g. can't, isn't, no). Note that you need to obtain necessary licence for commercial use.

First, plot the number of positive words and negative word in each statement, take the difference for the net sentiment. The positive word count and negative count are highly correlated and the average is on the positive side.



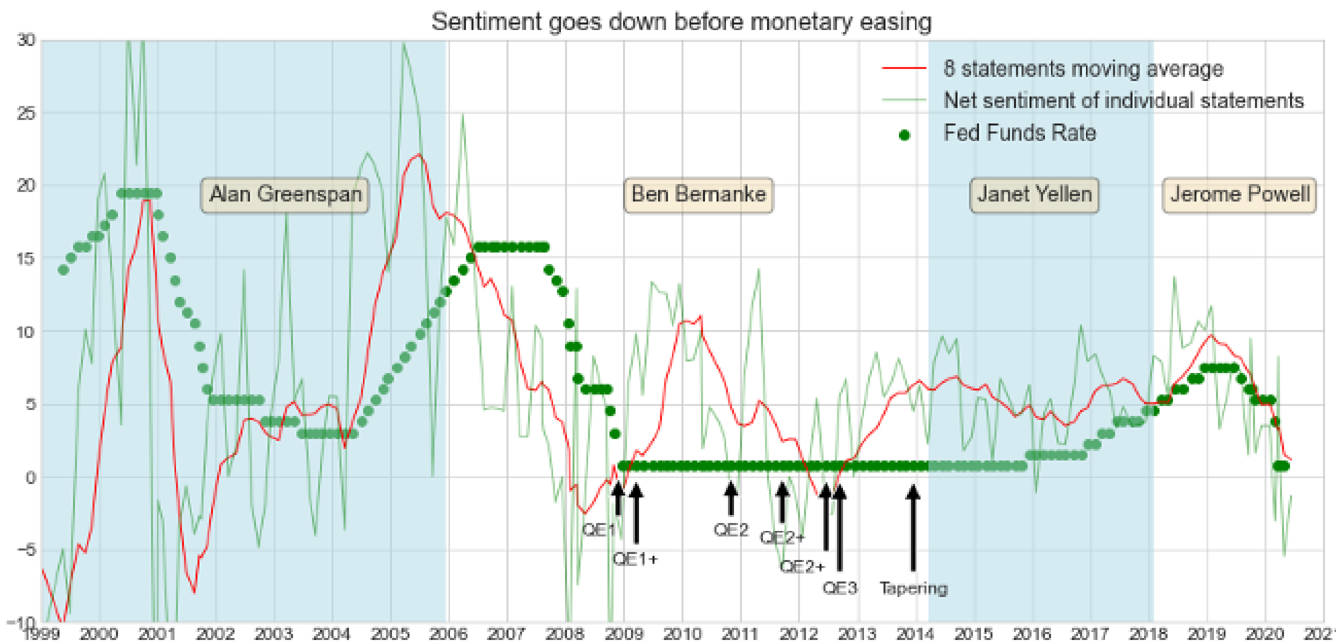
Sentiment over the years (Created by Author)

Next, apply the moving average to the net sentiment to see the trend plot with recession period. The moving average of 16 statements sentiment mostly goes down during economic uncertainty. This suggests the direction of net sentiment correlates with macro economics to some extent, while it would be too noisy to use for prediction at each FOMC meeting.



Moving Average of the sentiment and Recession (Created by Author)

Also check the moving average of net sentiment with actual FED Rate decisions at each FOMC meeting. There's a certain correlation with FED target rate, but it will not be easy to see during the Financial Crisis where the rate was at Effective Lower Boundary and quantitative easing was taken place. I treated QE announcement as a lowering rate event.



Moving Average of the sentiment and Monetary Policy (Created by Author)

Here is some sample code to generate this graph in matplotlib:

```
# Speaker window
Greenspan = np.logical_and(Data.index > '1987-08-11', Data.index < '2006-01-31')
Bernanke = np.logical_and(Data.index > '2006-02-01', Data.index < '2014-01-31')
Yellen = np.logical_and(Data.index > '2014-02-03', Data.index < '2018-02-03')
Powell = np.logical_and(Data.index > '2018-02-05', Data.index < '2022-02-05')
Speaker = np.logical_or.reduce((Greenspan, Yellen))

# Moving Average
Window = 8
NetSentMA = NetSentimentNorm.rolling(Window).mean()

# Prepare plot
fig, ax = plt.subplots(figsize=(15,7))
```

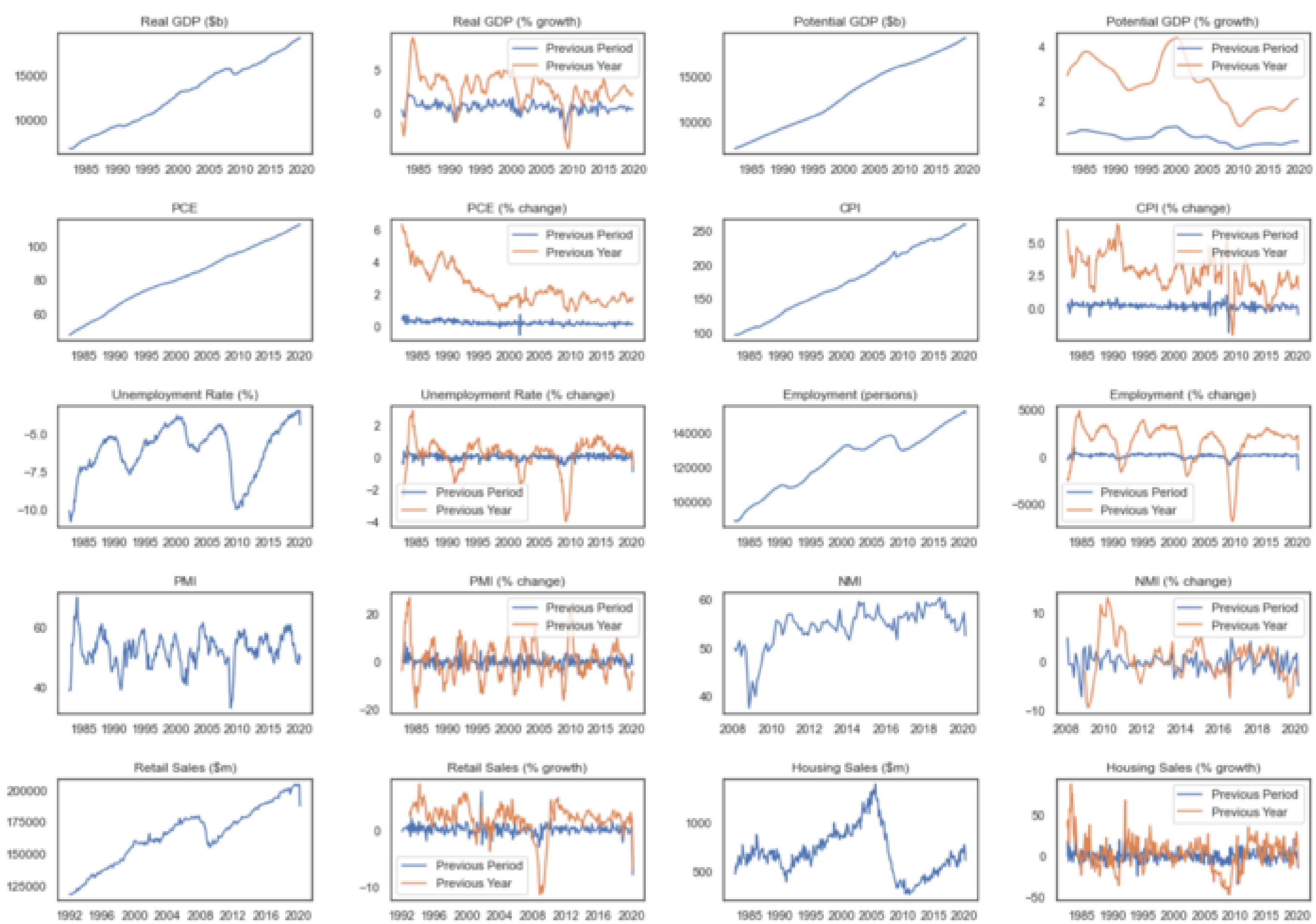
```
plt.title('Sentiment goes down before monetary easing', fontsize=16)

# Plotting Data
ax.scatter(Data.index, Data['Rate']*3, c = 'g')
ax.plot(Data.index, NetSentMA, c = 'r', linewidth= 1.0)
ax.plot(Data.index, NetSentimentNorm,  c = 'green', linewidth= 1, alpha = 0.5)
ax.legend([str(str(Window) + ' statements moving average'),
           'Net sentiment of individual statements',
           'Fed Funds Rate'], prop={'size': 14}, loc = 1)

# Format X-axis
import matplotlib.dates as mdates
years = mdates.YearLocator()   # every year
months = mdates.MonthLocator()  # every month
years_fmt = mdates.DateFormatter('%Y')

ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(years_fmt)
ax.xaxis.set_minor_locator(months)

# Set X-axis and Y-axis range
datemin = np.datetime64(Data.index[18], 'Y')
datemax = np.datetime64(Data.index[-1], 'Y') + np.timedelta64(1, 'Y')
ax.set_xlim(datemin, datemax)
ax.set_ylim(-10,30)

# Format the coords message box
ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
ax.grid(True)
ax.tick_params(axis='both', which='major', labelsize=12)

# Fill speaker
import matplotlib.transforms as mtransforms
trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
theta = 0.9
ax.fill_between(Data.index, 0, 10, where = Speaker, facecolor='lightblue', alpha=0.5, transform=

# Add text
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
ax.text(0.13, 0.75, "Alan Greenspan", transform=ax.transAxes, fontsize=14, verticalalignment='to
ax.text(0.46, 0.75, "Ben Bernanke", transform=ax.transAxes, fontsize=14, verticalalignment='top'
ax.text(0.73, 0.75, "Janet Yellen", transform=ax.transAxes, fontsize=14, verticalalignment='top'
ax.text(0.88, 0.75, "Jerome Powell", transform=ax.transAxes, fontsize=14, verticalalignment='top

# Add annotations
```

```python
# Add annotations
arrow_style = dict(facecolor='black', edgecolor='white', shrink=0.05)
ax.annotate('QE1', xy=('2008-11-25', 0), xytext=('2008-11-25', -4), size=12, ha='right', arrowpr
ax.annotate('QE1+', xy=('2009-03-18', 0), xytext=('2009-03-18', -6), size=12, ha='center', arrow
ax.annotate('QE2', xy=('2010-11-03', 0), xytext=('2010-11-03', -4), size=12, ha='center', arrowp
ax.annotate('QE2+', xy=('2011-09-21', 0), xytext=('2011-09-21', -4.5), size=12, ha='center', arr
ax.annotate('QE2+', xy=('2012-06-20', 0), xytext=('2012-06-20', -6.5), size=12, ha='right', arro
ax.annotate('QE3', xy=('2012-09-13', 0), xytext=('2012-09-13', -8), size=12, ha='center', arrowp
ax.annotate('Tapering', xy=('2013-12-18', 0), xytext=('2013-12-18', -8), size=12, ha='center', a

plt.show()
```

cb_sentiment_with_fedrate.py hosted with ♡ by GitHub view raw

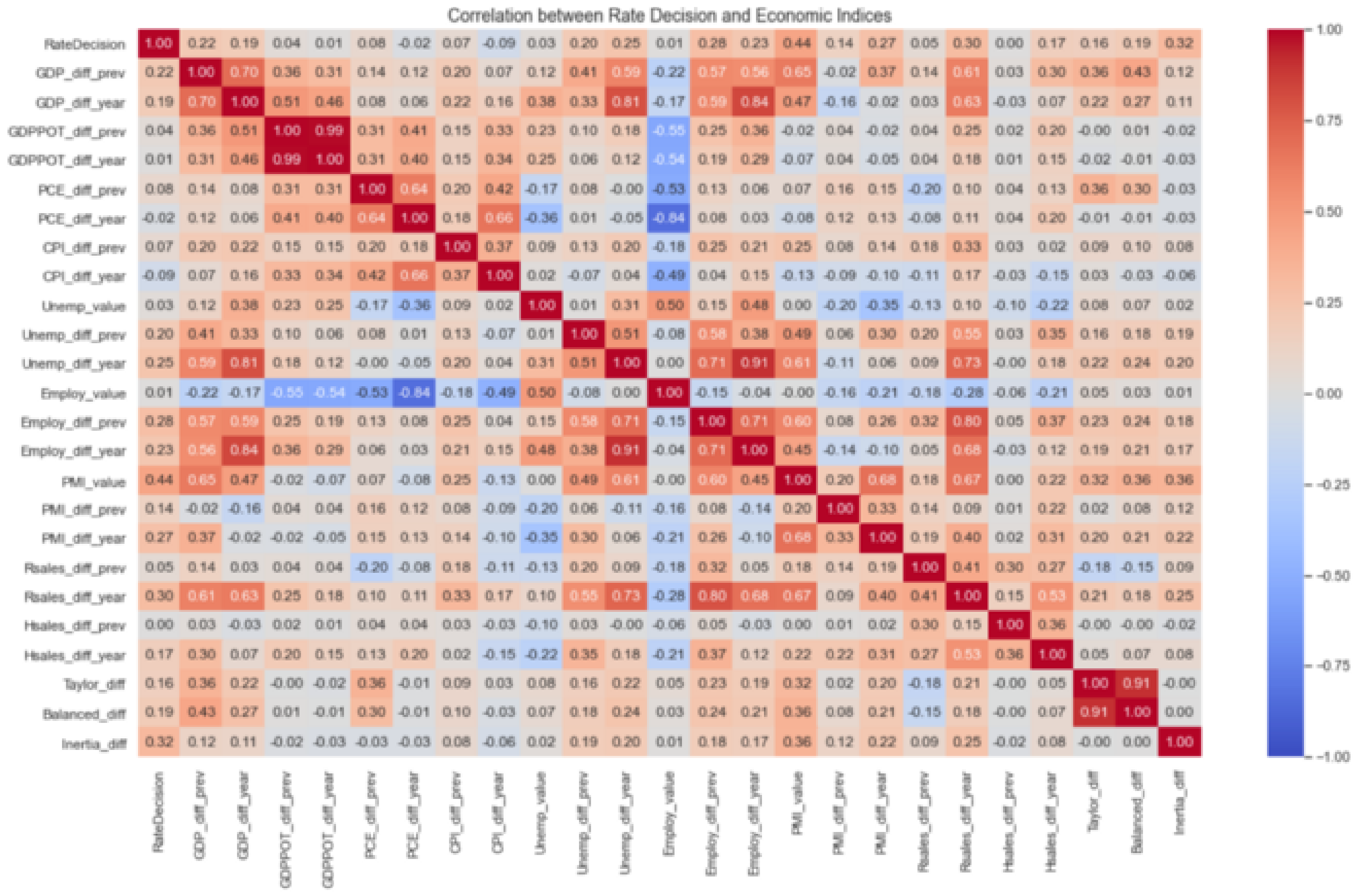
4. Pre-processing Economic Index

When FOMC decides the monetary policy, the difference from previous figure is also important. For each index, take difference from the previous period and the same period of the previous year for all the indices.



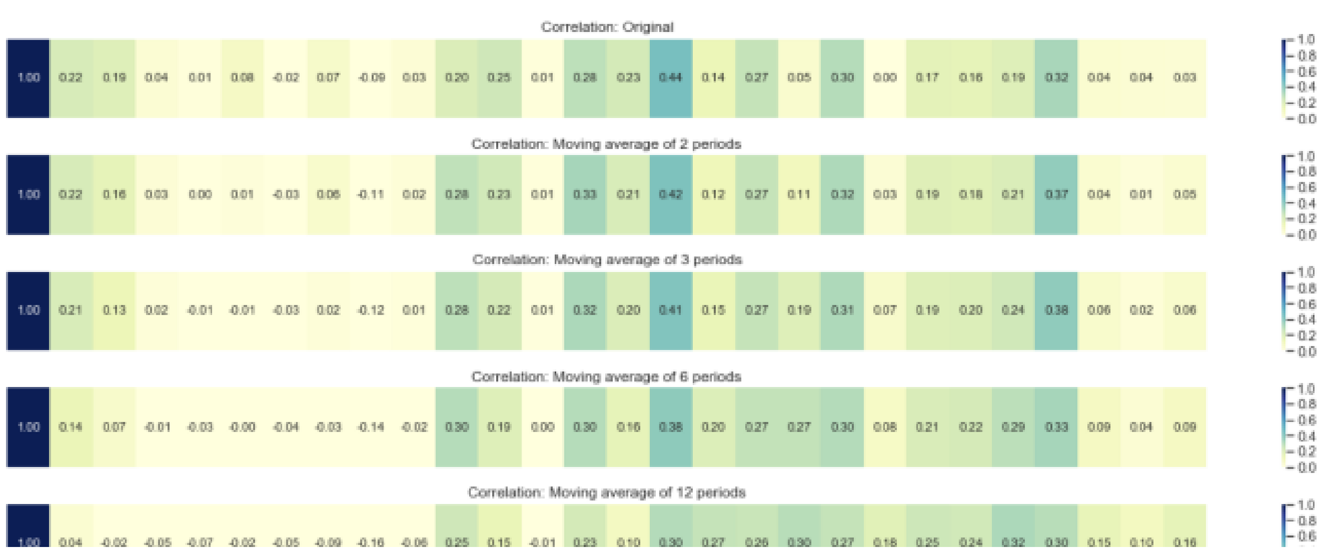
Economic Indices (Created by Author)

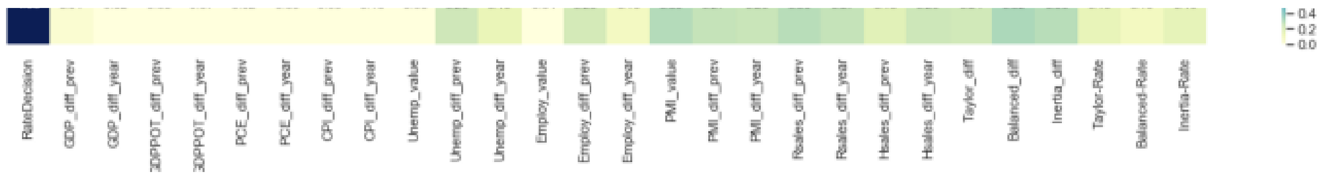
First check the correlation between FED Rate decision and economic indices using `seaborn.heatmap()` . PMI has higher correlation, while CPI has little to do with the rate decision.



Correlation between economic indices and rate decision at top row (Created by Author)

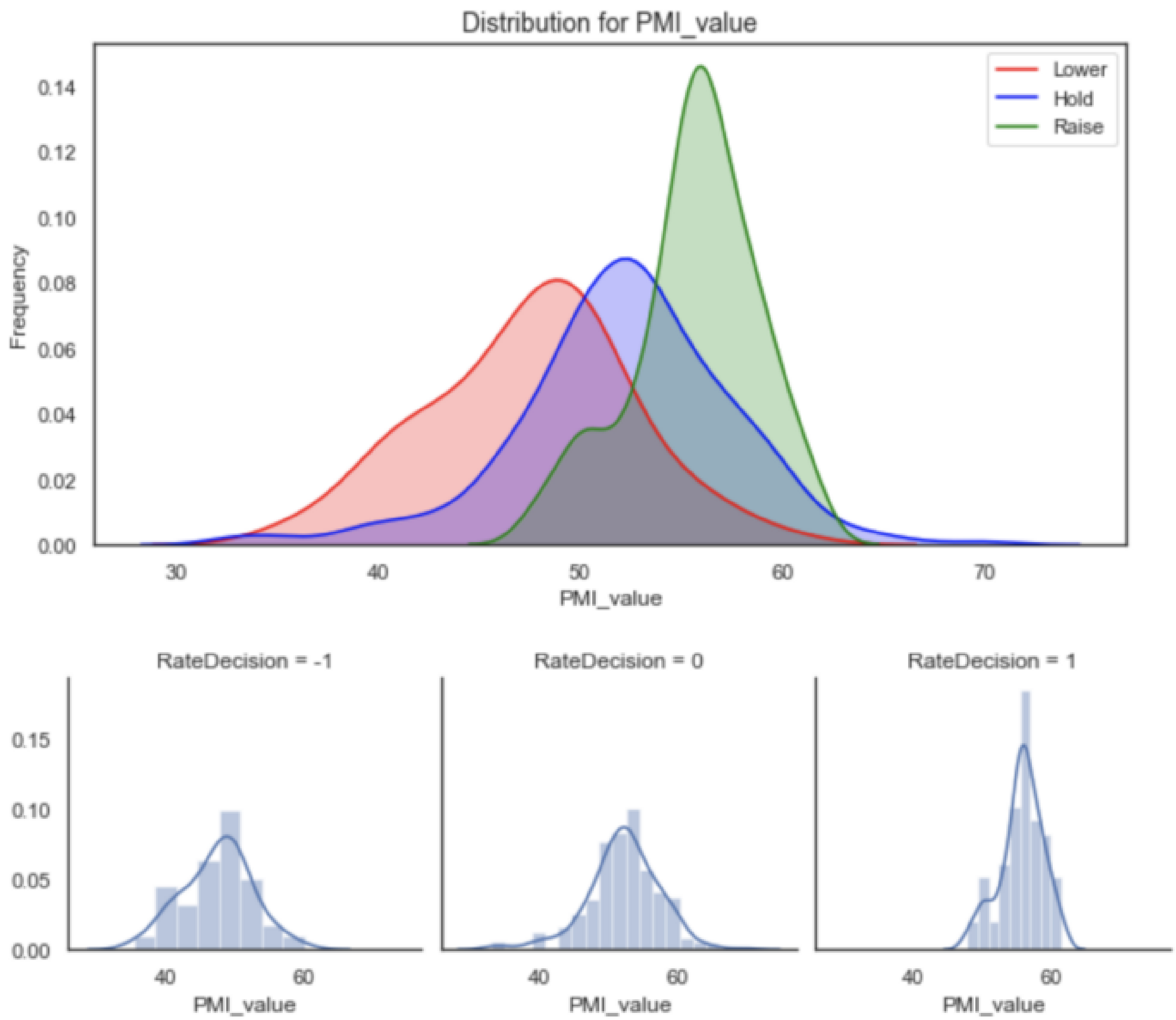
Then, calculate the moving averages of these numbers and check the correlation with the rate decision and select highly correlated fields as features.





Correlation for moving averages (Created by Author)

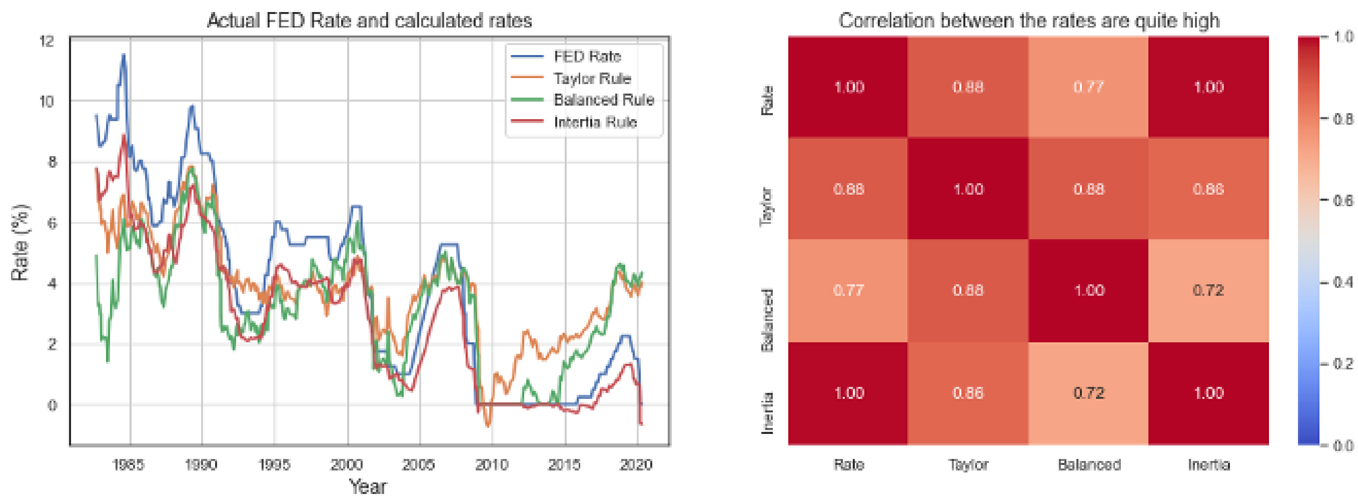
For example, PMI value is one of input feature candidates. `seaborn.kdeplot()` provides nice distribution plot.



Distribution of PMI values per the next Rate Decision (Created by Author)

As a part of feature engineering, calculate taylor rules and see whether the first derivatives and difference from FED rate could be used. FED has released how policymakers use economic indices data on their website. Here, calculate Taylor Rule, Balanced-approach Rule, and Inertial Rule from raw data.

The result looks to match with their publication and the correlation between these theoretical rates and actual FED Rates is quite high.



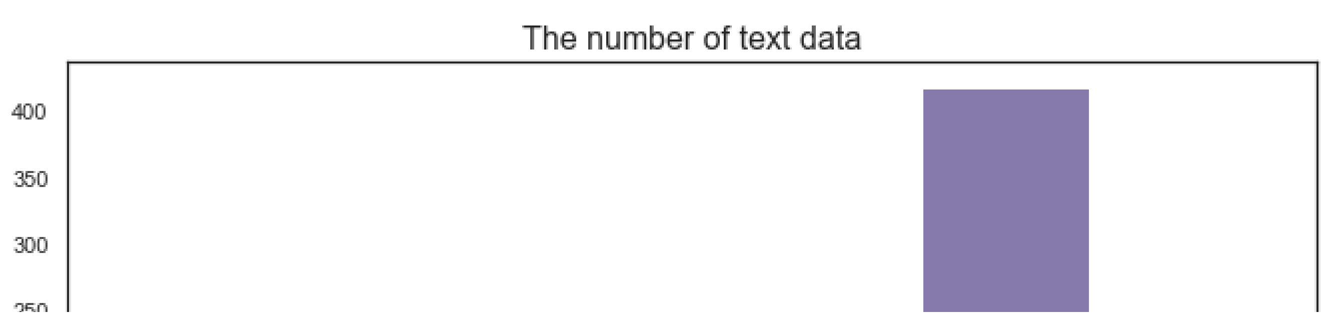
Theoretical Rates (Created by Author)

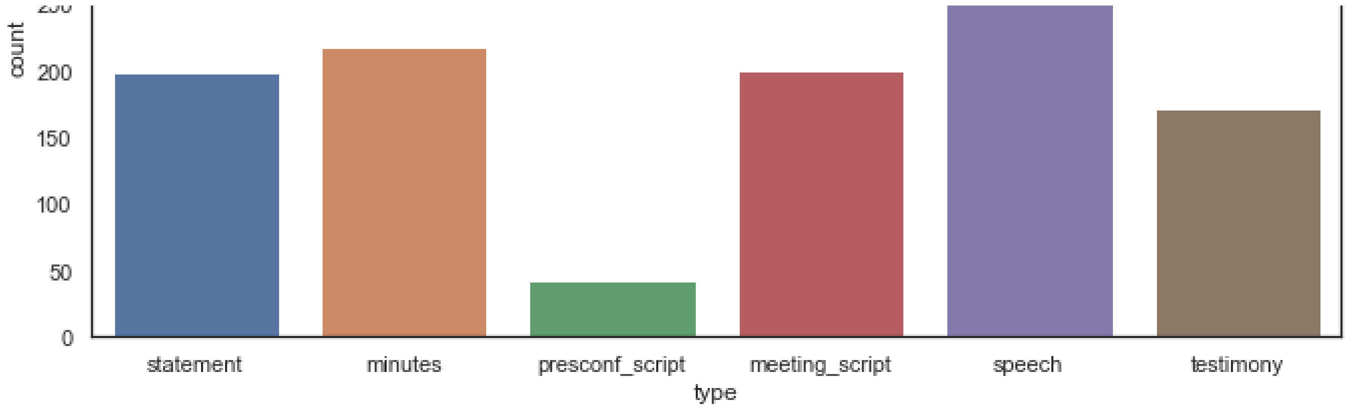
Do not forget to check the missing values and impute the values by 0 or mean/median for whichever makes sense. Also drop some of the records that do not have enough input data or missing output labels. A machine cannot learn from partially missing inputs as it anyway needs to make a guess on what the missing data mean.

The latest figure available at each FOMC meeting timing is used as the fundamental inputs to the decision and add textual data as additional inputs to see if the prediction can be improved.

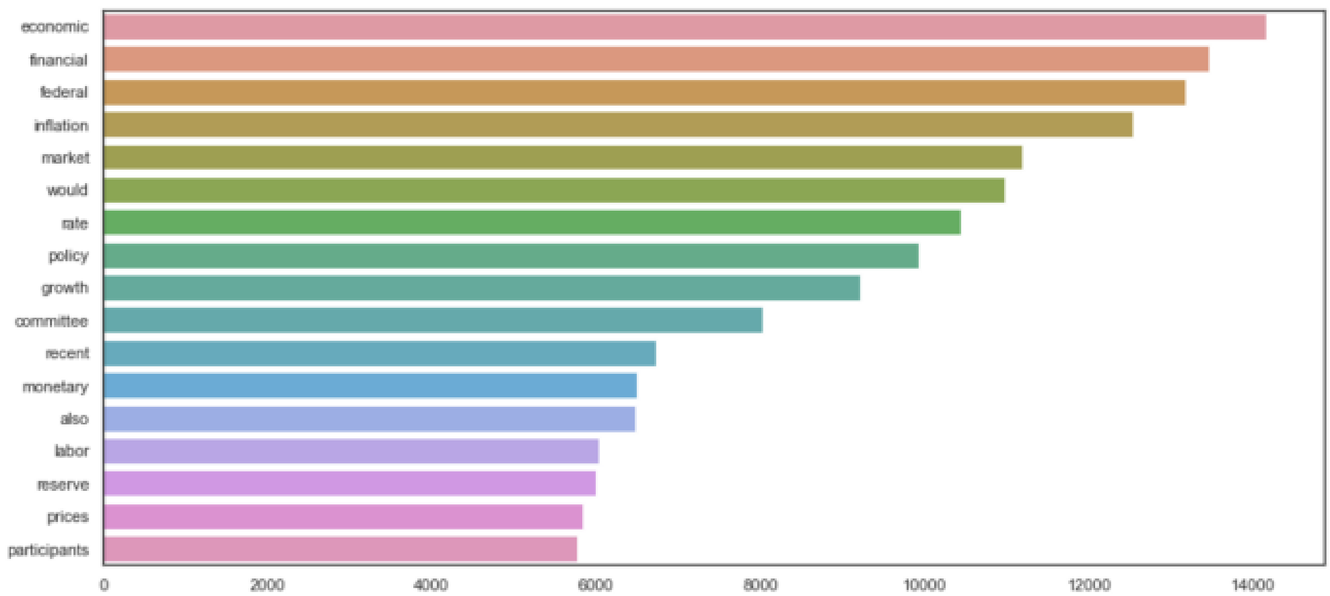
5. Pre-processing Text Data

There are around 200 decisions over the last two decades and a half. Depending on the models some inputs cannot be used due to missing data or available timing.





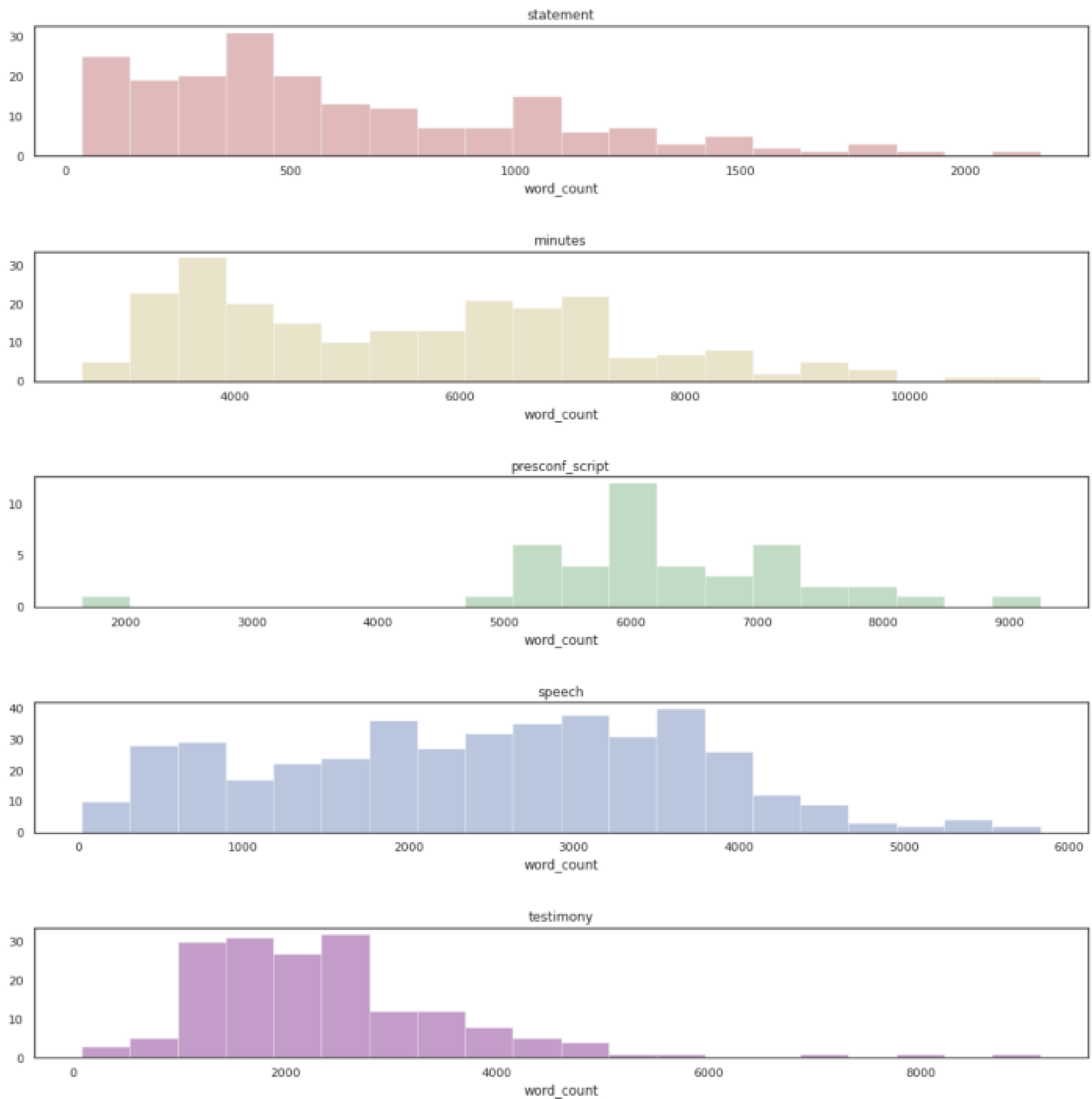
The input document counts per doc type (Created by Author)



Most Frequent Words (Created by Author)

Word Cloud (Created by Author)

One of common issues you may face during text processing is how to handle long text in machine learning. Most of the neural net based algorithms are not capable to analyse such long texts like 10,000 words — 500 at maximum. Most of our input texts are too long to analyse as a whole document.



The number of words in each sentence (Created by Author)

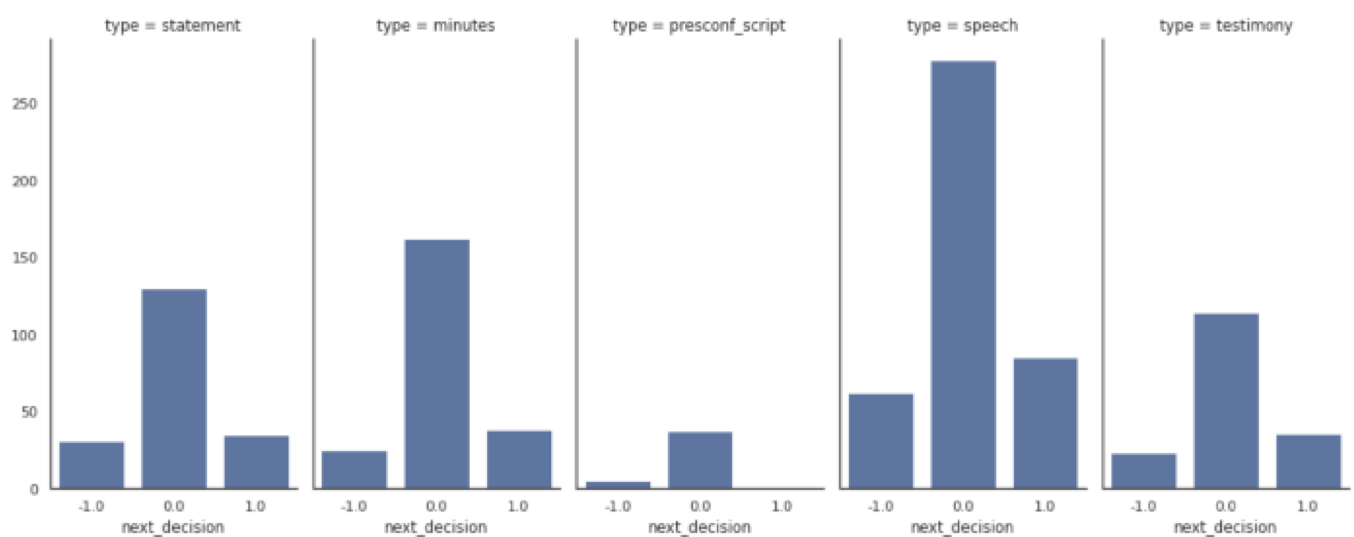
Typical solutions to this problem is either to use other algorithms such as jaccard/cosine similarity on the document vectors or to find a way to split the long text or to use some techniques to shorten such as text summarisation.

One simple solution used here is the text split technique to split the text by the number of words (e.g. 200 words with overlapping of 50 words) as shown below. This is a simple automated way but easily lose the context because the extracted 200 words may or may not contain relevant text and even off the topic.

```
def get_split(text, split_len=200, overlap=50):
    '''
    Returns a list of split text of $split_len with overlapping of $overlap.
    Each item of the list will have around split_len length of text.
    '''
    l_total = []
    words = re.findall(r'\b([a-zA-Z]+n\'t|[a-zA-Z]+\'s|[a-zA-Z]+)\b', text)

    if len(words) < split_len:
        n = 1
    else:
        n = (len(words) - overlap) // (split_len - overlap) + 1

    for i in range(n):
        l_parcial = words[(split_len - overlap) * i: (split_len - overlap) * i + split_len]
        l_total.append(" ".join(l_parcial))
    return l_total

def get_split_df(df, split_len=200, overlap=50):
    '''
    Returns a dataframe which is an extension of an input dataframe.
    Each row in the new dataframe has less than $split_len words in 'text'.
    '''
    split_data_list = []

    for i, row in tqdm(df.iterrows(), total=df.shape[0]):
        #print("Original Word Count: ", row['word_count'])
        text_list = get_split(row["text"], split_len, overlap)
        for text in text_list:
            row['text'] = text
            #print(len(re.findall(r'\b([a-zA-Z]+n\'t|[a-zA-Z]+\'s|[a-zA-Z]+)\b', text)))
            #row['word_count'] = len(re.findall(r'\b([a-zA-Z]+n\'t|[a-zA-Z]+\'s|[a-zA-Z]+)\b', t
            split_data_list.append(list(row))
```

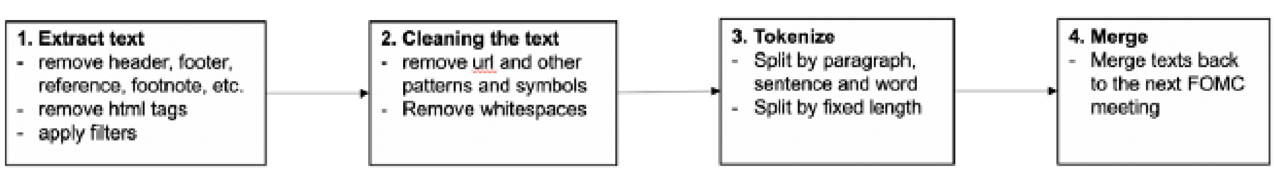
```
    split_df = pd.DataFrame(split_data_list, columns=df.columns)

    return split_df
```


Another issues is data imbalance — in this example, rate decision is “hold” for more than 60% chance and available decisions are only ~200 as the meeting is taken place eight times an year. Without having enough data, machine learning can easily overfit to the training data.



The number of input documents by document type (Created by Author)

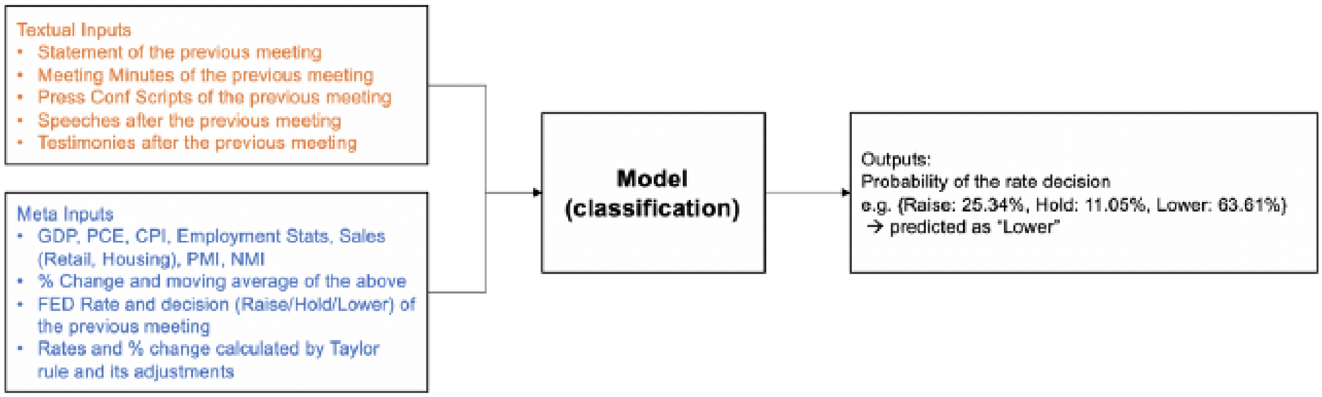
The text data processing have been done in the following manner as is often the case for NLP.



NLP Pipeline (Created by Author)

6. Build and Train ML Models

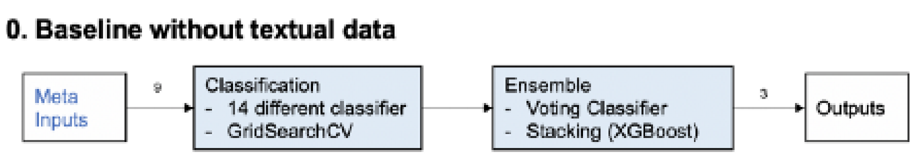
At high-level, the model takes textual inputs and meta inputs to predict three classes: Raise, Hold or Lower as follows. The point is how to combine textual input with numerical inputs and there are different ways to implement it.



Overall process flow (Created by Author)

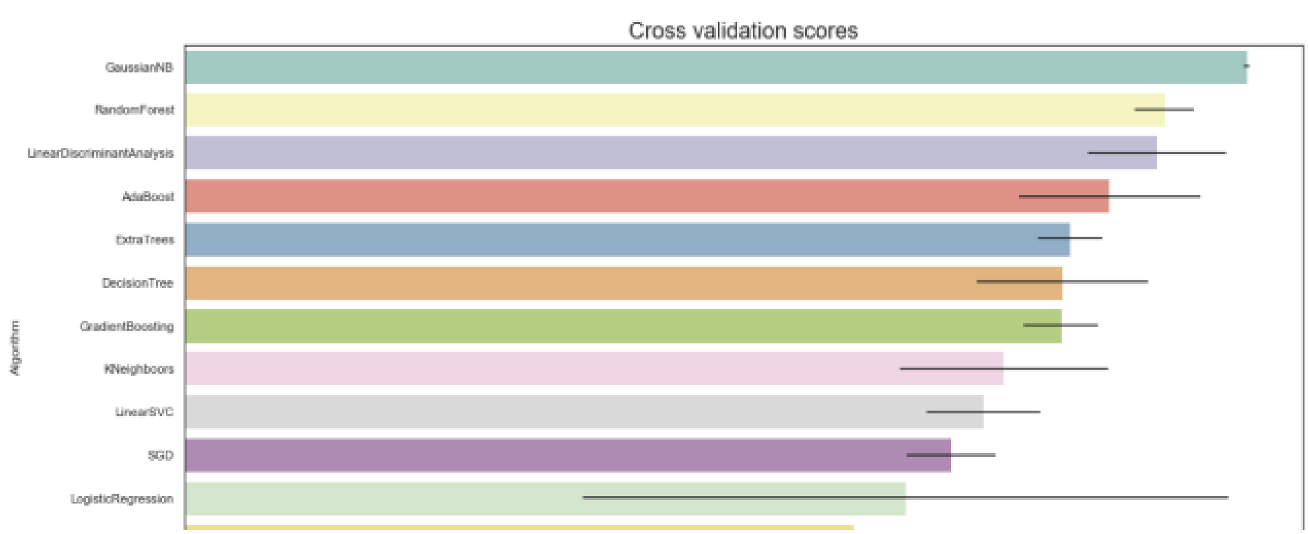
Here built the following six models in addition to the baseline model. All the models are built and trained in pytorch and the source codes are available in Github repo, of which some parts are extracted in this post.

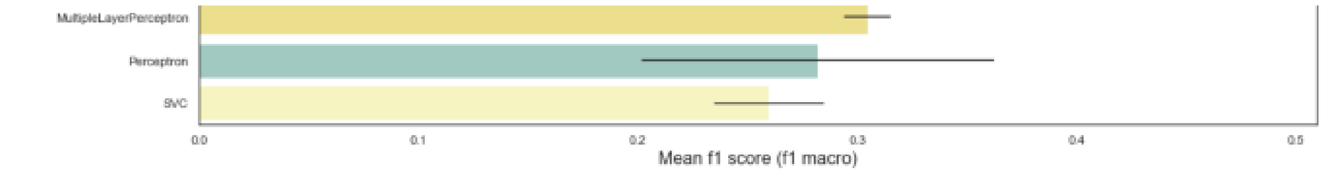
0. Baseline Model



Baseline process flow (Created by Author)

This does not use textual inputs but just take meta inputs. 14 different classifiers with the default parameters are compared first to grab the baseline performance quickly .



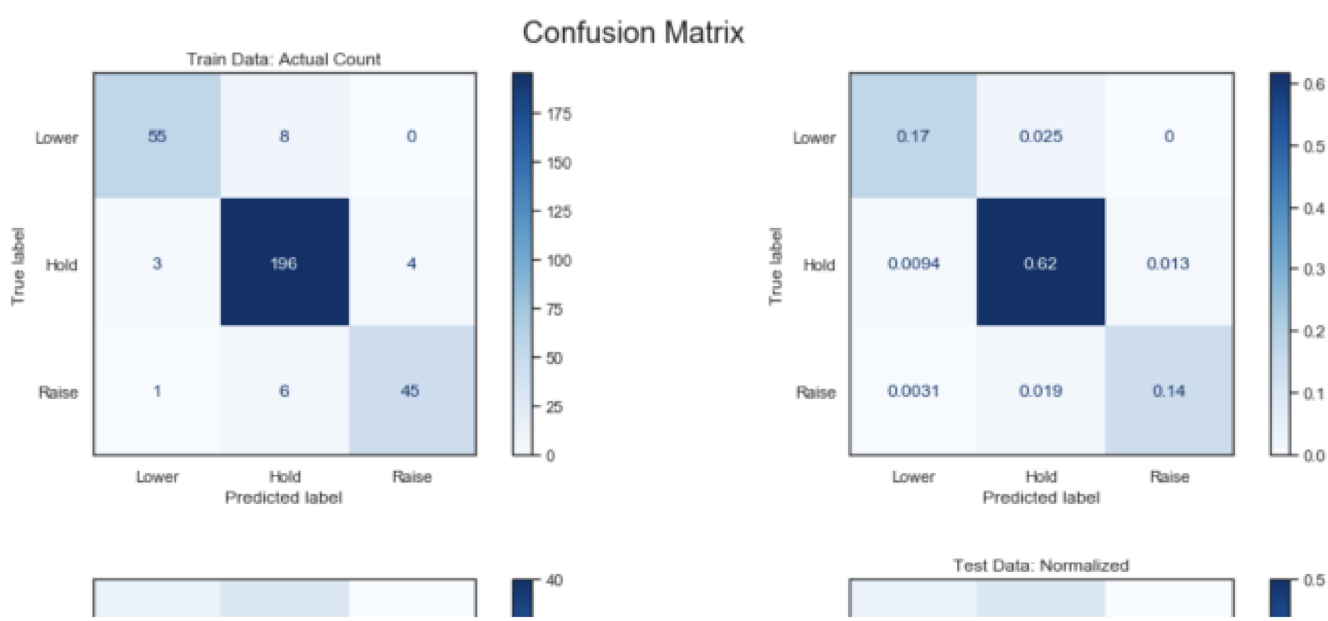


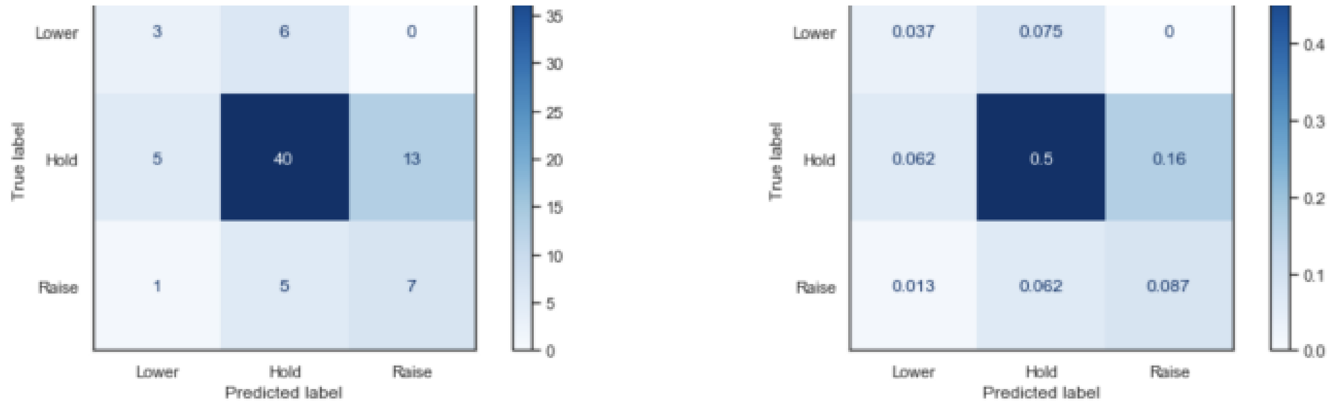
F1 Scores by 14 different classifiers (Created by Author)

Then, apply RandomizedSearchCV and GridSearchCV to find optimal hyper parameters and decided to use Random Forest as a base because it produces the best result with reasonable feature importance. StratifiedKFold is used for the cross validation.

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:   15.1s finished
[2020-06-19 06:49:12,703][INFO] ## Training - acc: 0.93081761, f1: 0.91396033
[2020-06-19 06:49:12,705][INFO] ## Test - acc: 0.62500000, f1: 0.49717357

Best Score:  0.4715680941564336
Best Param:  {'bootstrap': False, 'criterion': 'gini', 'max_depth': None, 'max_features': 8, 'min_samples_leaf': 3,
'min_samples_split': 2, 'n_estimators': 31}
```

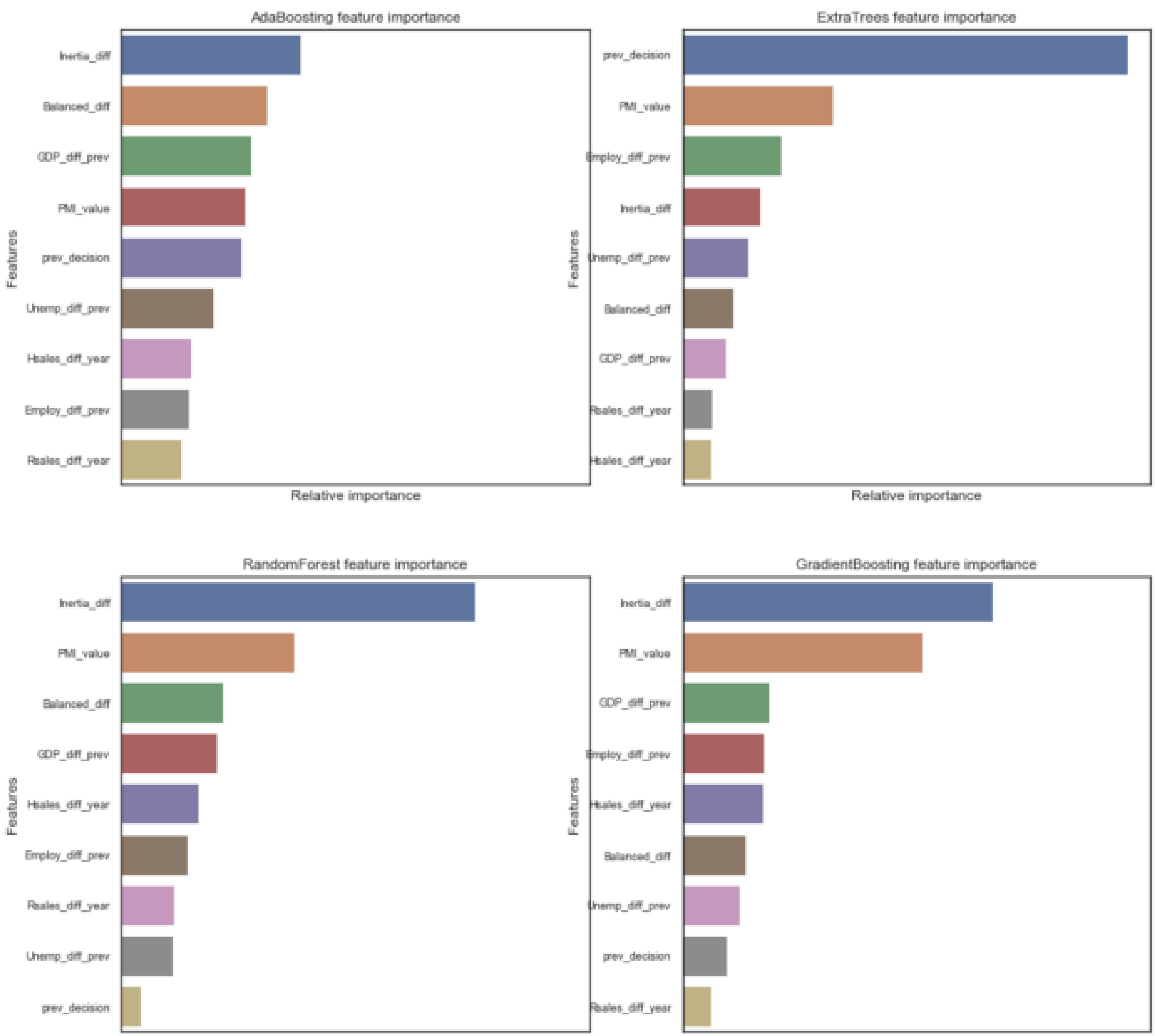
Grid Search CV Result (Created by Author)





Confusion Matrix (Created by Author)

Learning curve tells that the model overfits to training data and more data could potentially improve the performance. Looking at the confusion matrix, it is failing to predict “Lower” and “Raise” events.



Feature Importance (Created by Author)

Looking at the confusion matrix, it is failing to predict “Lower” and “Raise” events.

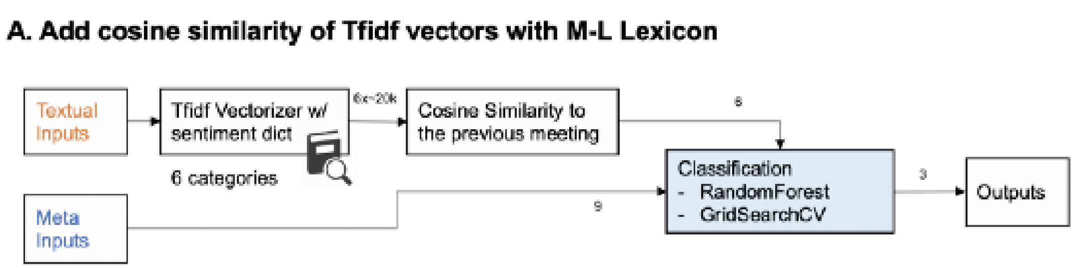
Ensemble and Stacking is also performed but did not improve the performance as the fundamental issue here is a lack of sufficient data.

```
### Voting Classifier
# Voting by best estimators with "soft" to take all the probability into account
voting_best = VotingClassifier(estimators=[('adac', ada_best),
                                           ('extc', ext_best),
                                           ('rfc', rf_best),
                                           ('gbc', gb_best),
                                           ('svmc', svm_best)], voting='soft', n_jobs=-1)
# Fit
voting_best.fit(X_train, Y_train)

# Predict
voting_pred_train = voting_best.predict(X_train)
voting_pred_test = voting_best.predict(X_test)

### XGBoost
# Get out of fold estimation for both training and test data
def get_oof(clf, x_train, y_train, x_test):
    #Set parameters for ensembling
    n_train = x_train.shape[0]
    n_test = x_test.shape[0]
    oof_train = np.zeros((n_train,))
    oof_test = np.zeros((n_test,))
    oof_test_skf = np.empty((n_fold, n_test))

    for i, (train_index, test_index) in enumerate(kfold.split(y_train, y_train)):
        x_tr = x_train[train_index]
        y_tr = y_train[train_index]
        x_te = x_train[test_index]

        clf.fit(x_tr, y_tr)

        oof_train[test_index] = clf.predict(x_te)
        oof_test_skf[i, :] = clf.predict(x_test)

    oof_test[:] = oof_test_skf.mean(axis=0)
```

```python
    oof_test[:] = oof_test_skf.mean(axis=0)
    return oof_train.reshape(-1,1), oof_test.reshape(-1, 1)

# Create OOF by the best estimators
ada_oof_train, ada_oof_test = get_oof(ada_best, X_train, Y_train, X_test) # AdaBoost
ext_oof_train, ext_oof_test = get_oof(ext_best, X_train, Y_train, X_test) # Extra Trees
rf_oof_train, rf_oof_test = get_oof(rf_best, X_train, Y_train, X_test) # Random Forest
gb_oof_train, gb_oof_test = get_oof(gb_best, X_train, Y_train, X_test) # Gradient Boost
svmc_oof_train, svmc_oof_test = get_oof(svm_best, X_train, Y_train, X_test) # Support Vector Cla

# Stacking
X_train_xgb = np.concatenate((ada_oof_train, ext_oof_train, rf_oof_train, gb_oof_train, svmc_oof
X_test_xgb = np.concatenate((ada_oof_test, ext_oof_test, rf_oof_test, gb_oof_test, svmc_oof_test

# Fit
import xgboost as xgb
gbm = xgb.XGBClassifier(n_estimator=2000, max_depth=4, min_child_weight=2, gamma=0.9,
                        subsample=0.8, colsample_bytree=0.8, objective='binary:logistic',
                        nthread=-1, scale_pos_weight=1).fit(X_train_xgb, Y_train)
# Predict
gbm_pred_train = gbm.predict(X_train_xgb)
gbm_pred_test = gbm.predict(X_test_xgb)
```

cb_baseline_ensemble.py hosted with ♡ by GitHub view raw

A. Cosine Similarity



Cosine Similarity Process flow (Created by Author)

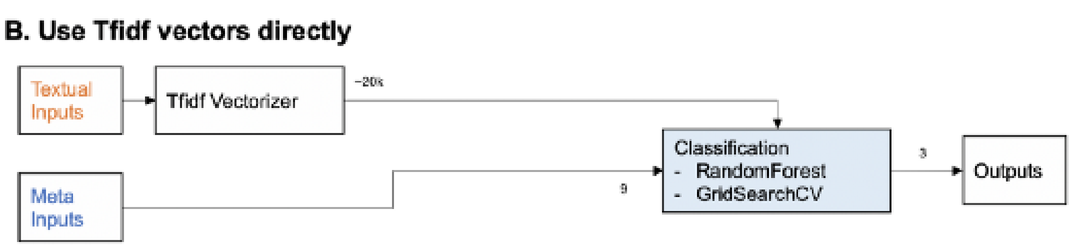
The texts are vectorised by Tfidf using Loughran-McDonald dictionary, which is used in preliminary analysis, and calculate the cosine similarity between two consecutive meetings. This value is the degree of change in the text direction (i.e. cosine of vectors), which may indicate the policy change. This is then combined with economic indices used in the baseline model.

```
# Function to lemmatize a word
def lemmatize_word(word):
    wnl = nltk.stem.WordNetLemmatizer()
    return wnl.lemmatize(wnl.lemmatize(word, 'n'), 'v')

# Function to tokenize text in a DataFrame
def tokenize_df(df, col='text'):
    tokenized = []
    wnl = nltk.stem.WordNetLemmatizer()
    for text in tqdm(df[col]):
        # Filter alphabet words only and non stop words, make it loser case
        words = [word.lower() for word in word_tokenize(text) if ((word.isalpha()==1) & (word no
        # Lemmatize words
        tokens = [lemmatize_word(word) for word in words]
        tokenized.append(tokens)
    return tokenized

# Function to create Tfidf Vector
from sklearn.feature_extraction.text import TfidfVectorizer
def get_tfidf(sentiment_words, docs):
    vectorizer = TfidfVectorizer(analyzer='word', vocabulary=sentiment_words)
    tfidf = vectorizer.fit_transform(docs)
    features = vectorizer.get_feature_names()

    return tfidf.toarray()

# Function to calculate Cosine Similarity
from sklearn.metrics.pairwise import cosine_similarity
def get_cosine_similarity(tfidf_matrix):
    return [cosine_similarity(u.reshape(1,-1), v.reshape(1,-1))[0][0].tolist() for u, v in zip(t

# Tokenize
tokenized = tokenize_df(train_df)
docs = [" ".join(words) for words in tokenized]

# Create vocab
all_words = [word for text in tokenized for word in text]
counts = Counter(all_words)
bow = sorted(counts, key=counts.get, reverse=True)
vocab = {word: ii for ii, word in enumerate(counts, 1)}
id2vocab = {v: k for k, v in vocab.items()}

# Create token id list
token_ids = [[vocab[word] for word in text_words] for text_words in tokenized]

```

```python
# Lemmertize sentiment word list as well
lemma_sentiment_df = lmdict_df.copy(deep=True)
lemma_sentiment_df['word'] = [lemmatize_word(word) for word in lemma_sentiment_df['word']]
lemma_sentiment_df = sentiment_df.drop_duplicates('word') # Drop duplicates
lemma_sentiments = list(lemma_sentiment_df['sentiment'].unique()) # Sentiment list

# Create Tfidf dictionary
sentiment_tfidf = {
   sentiment: get_tfidf(lemma_sentiment_df.loc[lemma_sentiment_df['sentiment'] == sentiment]['wo
   for sentiment in lemma_sentiments}

# Create Cosine Similarity dictionary
cosine_similarities = {
   sentiment_name: get_cosine_similarity(sentiment_values)
   for sentiment_name, sentiment_values in sentiment_tfidf.items()}

# Add to DataFrame
for sentiment in lemma_sentiments:
    # Add 0 to the first element as there is no comparison available to a previous value
    cosine_similarities[sentiment].insert(0, 0)
    train_df['cos_sim_' + sentiment] = cosine_similarities[sentiment]
```

cb_cosine_similarity.py hosted with ♡ by GitHub view raw

B. Tfidf



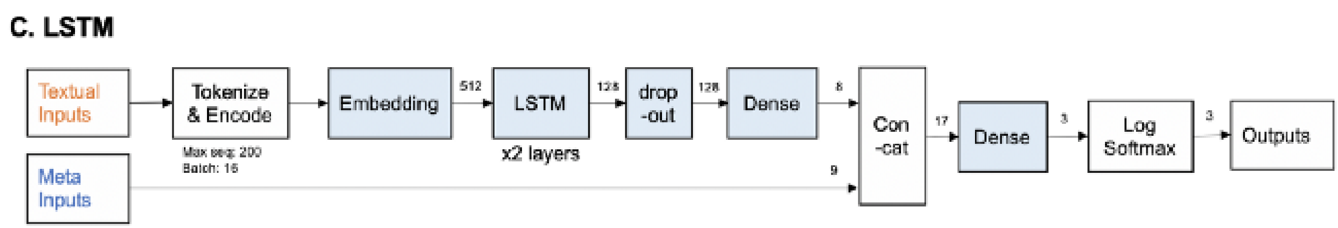
Tfidf Process flow (Created by Author)

Instead of cosine similarity, use the Tfidf vector itself as input. This would only work if the Tfidf vector directly holds meaningful information on the rate change. Using the tokenized text in the previous step, concatenate the Tfidf Vector with Non-textual inputs by FunctionTransfomer.

```python
import scipy
from sklearn.preprocessing import FunctionTransformer
```

```python

# Create Function Transformer to use Feature Union
def get_numeric_data(x):
    return [record[:-2].astype(float) for record in x]

def get_text_data(x):
    return [record[-1] for record in x]

transfomer_numeric = FunctionTransformer(get_numeric_data)
transformer_text = FunctionTransformer(get_text_data)

# Create Vocabrary from L-M Dict
vocabulary=sentiment_dict['Negative']+sentiment_dict['Positive']

# Create a pipeline to concatenate Tfidf Vector and economic indices
pipeline = Pipeline([
    ('features', FeatureUnion([
            ('numeric_features', Pipeline([
                ('selector', transfomer_numeric)
            ])),
             ('text_features', Pipeline([
                ('selector', transformer_text),
                ('vec', TfidfVectorizer(analyzer='word'))
            ]))
         ])),
    ('clf', RandomForestClassifier())
])

# Perform Grid Search
param_grid = {'clf__n_estimators': np.linspace(1, 60, 10, dtype=int),
              'clf__min_samples_split': [3, 10],
              'clf__min_samples_leaf': [3],
              'clf__max_features': [7],
              'clf__max_depth': [None],
              'clf__criterion': ['gini'],
              'clf__bootstrap': [False]}

rf_model = GridSearchCV(pipeline, param_grid=param_grid, cv=kfold, scoring=scoring, verbose=verb
                          refit=refit, n_jobs=-1, return_train_score=True)
rf_model.fit(X_train, Y_train)
rf_best = rf_model.best_estimator_
```

C. LSTM



LSTM Process flow (Created by Author)

LSTM (Long Short-Term Memory) is a popular RNN (Recurrent Neural Network) architecture that can hold long-term memory and short-term memory for sequence learning. There are a number of improved versions such as bidirectional LSTM while the one used here is a simple plain model. The output from the deep neural network is combined with economic indices after dropout and dense layer.

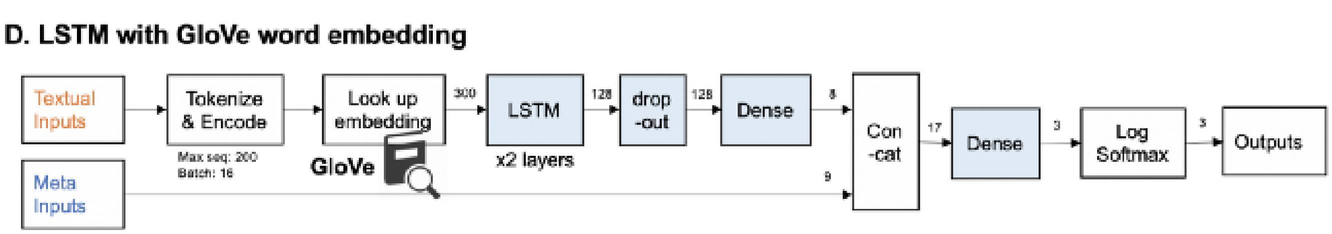
```
# Model taking meta (non-text) inputs as well
class TextClassifier(nn.Module):
    def __init__(self, vocab_size, embed_size, lstm_size, dense_size, meta_size, output_size, ls
        """
        Initialize the model
        """
        super().__init__()
        self.vocab_size = vocab_size
        self.embed_size = embed_size
        self.lstm_size = lstm_size
        self.output_size = output_size
        self.lstm_layers = lstm_layers
        self.dropout = dropout

        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, lstm_size, lstm_layers, dropout=dropout, batch_first=Fal
        self.dropout = nn.Dropout(0.2)
        self.fc1 = nn.Linear(lstm_size, dense_size)
        self.fc2 = nn.Linear(dense_size + meta_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def init_hidden(self, batch_size):
        """
        Initialize the hidden state
        """
        weight = next(self.parameters()).data
```

```python
        hidden = (weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_(),
                  weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_())

        return hidden

    def forward(self, nn_input_text, nn_input_meta, hidden_state):
        """
        Perform a forward pass of the model on nn_input
        """
        batch_size = nn_input_text.size(0)
        nn_input_text = nn_input_text.long()
        embeds = self.embedding(nn_input_text)
        lstm_out, hidden_state = self.lstm(embeds, hidden_state)
        # Stack up LSTM outputs, apply dropout
        lstm_out = lstm_out[-1,:,:]
        lstm_out = self.dropout(lstm_out)
        # Dense layer
        dense_out = self.fc1(lstm_out)
        # Concatinate the dense output and meta inputs
        concat_layer = torch.cat((dense_out, nn_input_meta.float()), 1)
        out = self.fc2(concat_layer)
        logps = self.softmax(out)

        return logps, hidden_state
```

cb_lstm_model.py hosted with ♡ by GitHub　　view raw

Then, separate the input to textual data and numeric data. The data loader is customised to yield the two types inputs. Training process is the same as usual case of processing text by LSTM.

D. LSTM+GloVe



LSTM + GloVe Process flow (Created by Author)

The previous LSTM model creates own word embedding but there's also pre-trained embedding. Here uses Global Vectors for Word Representation (GloVe) which was trained by wikipedia and gigaword (6B tokens). The idea is the pretrained word representation would boost the performance of the randomly initialised model.

```
# Use 6B 300d
glove_file = 'glove.6B.300d.pickle'
glove_path = glove_dir + glove_file

# Download Glove file if not exist
if not os.path.exists(glove_path):
    if not os.path.exists(glove_path):
        os.mkdir(glove_path)
    !wget -o ${glove_dir} http://nlp.stanford.edu/data/glove.6B.zip
    !unzip ${glove_dir}glove*.zip

    embedding_dict = {}

    with open(glove_dir + "glove.6B.300d.txt", 'r') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vectors = np.asarray(values[1:], 'float32')
            embedding_dict[word] = vectors
    f.close()

    pickle.dump(embedding_dict, open(glove_path, 'wb'))

# Open downloaded GloVe dict
glove_dict = pickle.load(open(glove_path, 'rb'))

# Set random weight for words that are not in the GloVe dict
weight_matrix = np.zeros((len(vocab), 300))
words_found = 0
for i, word in enumerate(vocab):
    try:
        weight_matrix[i] = glove_dict[word]
        words_found += 1
    except KeyError:
        weight_matrix[i] = np.random.normal(scale=0.6, size=(300,))

# Classifier definition
class GloveTextClassifier(nn.Module):
```

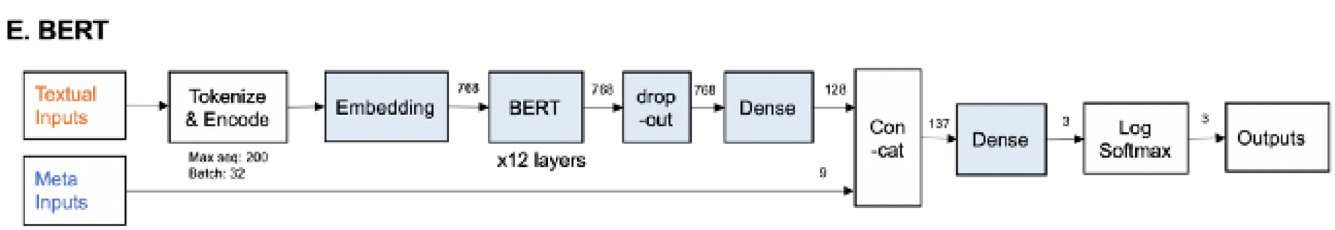
```
    def __init__(self, weight_matrix, lstm_size, dense_size, meta_size, output_size, lstm_layers
        """
        Initialize the model
        """
        super().__init__()
        vocab_size, embed_size = weight_matrix.shape
        self.lstm_size = lstm_size
        self.output_size = output_size
        self.lstm_layers = lstm_layers
        self.dropout = dropout

        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.embedding.load_state_dict({'weight': torch.tensor(weight_matrix)})
        self.embedding.weight.requires_grad = False
        self.lstm = nn.LSTM(embed_size, lstm_size, lstm_layers, dropout=dropout, batch_first=Fal
        self.dropout = nn.Dropout(0.2)
        self.fc1 = nn.Linear(lstm_size, dense_size)
        self.fc2 = nn.Linear(dense_size + meta_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def init_hidden(self, batch_size):
        """
        Initialize the hidden state
        """
        weight = next(self.parameters()).data
        hidden = (weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_(),
                  weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_())

        return hidden

    def forward(self, nn_input_text, nn_input_meta, hidden_state):
        """
        Perform a forward pass of the model on nn_input
        """
        batch_size = nn_input_text.size(0)
        nn_input_text = nn_input_text.long()
        embeds = self.embedding(nn_input_text)
        lstm_out, hidden_state = self.lstm(embeds, hidden_state)
        # Stack up LSTM outputs, apply dropout
        lstm_out = lstm_out[-1,:,:]
        lstm_out = self.dropout(lstm_out)
        # Dense layer
        dense_out = self.fc1(lstm_out)
        # Concatinate the dense output and meta inputs
        concat_layer = torch.cat((dense_out, nn_input_meta.float()), 1)
```

```
        concat_layer = torch.cat((dense_out, nn_input_meta.float()), 1)
        out = self.fc2(concat_layer)
        logps = self.softmax(out)

        return logps, hidden_state
```

cb_glove_lstm_model.py hosted with ♡ by GitHub   view raw

The training steps are the same as the previous LSTM model.

E. BERT



BERT Process flow (Created by Author)

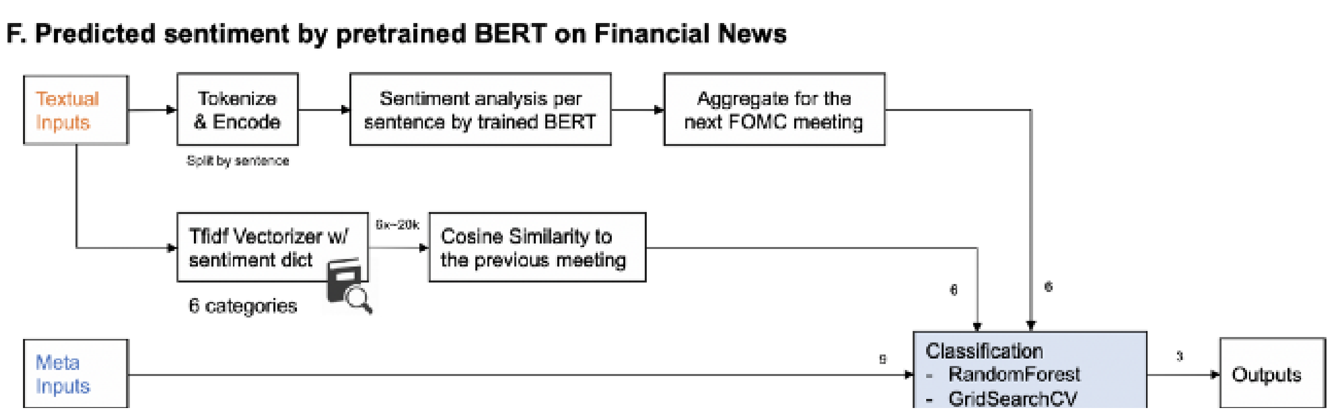
BERT, or Bidirectional Encoder Representations from Transformers, is a transformer based language model as opposed to RNN, published by Google Research in 2018. The model used here is pretrained BERT_BASE, which is a deep neural network with12 layers, 768 hidden units, 12 heads, resulting in 110M parameters and was trained on the Wikipedia and BooksCorpus. Apart from the model and BERT's own tokenisation, the rest of architecture stays the same as the LSTM based model above. Usually it would be sufficient to use `transformers.BertForSequenceClassification` for the model but here needs to create own definition to concatenate text with non-text inputs.

```
from transformers import BertTokenizer, BertModel

# Tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)

# Model definition
class BertTextClassifier(nn.Module):
    def __init__(self, hidden_size, dense_size, meta_size, output_size, dropout=0.1):
        """
        Initialize the model
        """
        super().__init__()
```

```python
        self.output_size = output_size
        self.dropout = dropout
        self.bert = BertModel.from_pretrained('bert-base-uncased',
                                              output_hidden_states=True,
                                              output_attentions=True)
        for param in self.bert.parameters():
            param.requires_grad = True
        self.weights = nn.Parameter(torch.rand(13, 1))
        self.dropout = nn.Dropout(dropout)
        self.fc1 = nn.Linear(hidden_size, dense_size)
        self.fc2 = nn.Linear(dense_size + meta_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input_ids, nn_input_meta):
        """
        Perform a forward pass of the model on nn_input
        """
        all_hidden_states, all_attentions = self.bert(input_ids)[-2:]
        batch_size = input_ids.shape[0]
        ht_cls = torch.cat(all_hidden_states)[:, :1, :].view(13, batch_size, 1, 768)
        atten = torch.sum(ht_cls * self.weights.view(13, 1, 1, 1), dim=[1, 3])
        atten = F.softmax(atten.view(-1), dim=0)
        feature = torch.sum(ht_cls * atten.view(13, 1, 1, 1), dim=[0, 2])
        # Dense layer
        dense_out = self.fc1(self.dropout(feature))
        # Concatinate the dense output and meta inputs
        concat_layer = torch.cat((dense_out, nn_input_meta.float()), 1)
        out = self.fc2(concat_layer)

        return out
```

cb_bert_model.py hosted with ♡ by GitHub — view raw

F. BERT + Pre-Sentiment Analysis

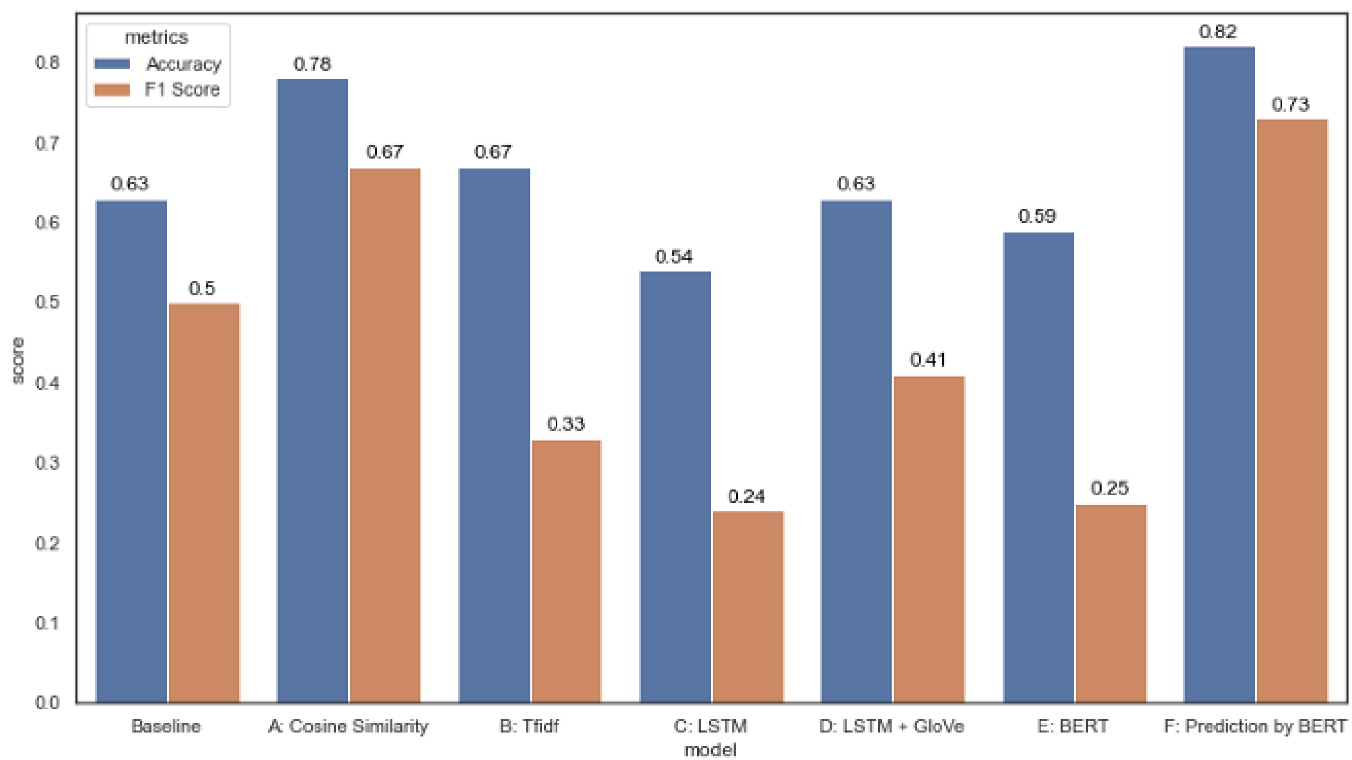


BERT + Pre-Sentiment Analysis Process flow (Created by Author)

Finally, I took another approach — instead of training the model directly on FOMC text, first train the model on other financial texts for sentiment analysis task. Then used the trained BERT model to analyse the sentiment of each sentence in FOMC text and calculate sentiment scores, which were then aggregated for each document and used as inputs to another ML model to predict the FOMC decision.

Result

The following shows the comparison of scores between the tested models. The deep neural network models performed poorly, which is basically due to lack of enough data to train these complex models. The last model with pre-sentiment analysis outperformed the other models but need to examine further if this improvement is with meaningful significance and consistency, and not due to just any additional inputs.



Performance Scores of each model (Created by Author)

Conclusion

We examined whether FOMC text data contains useful insight to predict the FED target rate decision (i.e. Raise, Hold or Lower) at the next FOMC meeting. We could observe some useful information in the text to predict FOMC decision better. However, we could not improve the text based prediction performance by Neural Network. This is partly because there are small number of test data to train with each text very long.

As a future work, there're two main areas to improve:

1. Tackle the lack of enough training data — The models have clearly overfitted to train samples and failed to generalise well, especially boosting algorithms are prone to overfitting. Hyperparameter tuning and imputation was considered there. In addition, configuring the model and splitting data to augment the training data by synthetic approach could potentially beneficial.
2. Improve input text quality — The input texts contain a lot of irrelevant paragraphs, which have nothing to to with FED target rate decision. For example, there are information about regulations, organisation structure and infrastructures. Filtering out less relevant inputs will improve the accuracy of the model prediction as well as training efficiency.

Note that the code in this post is just some extracts. Please refer to Github repo for the complete source code.
