MSML – 641 "Natural Language Processing"
Final Project

Prepared by: Ricardo Zambrano Duran
UMD iD: 115811614

I. Introduction and Problem Definition

In this section a brief summary about what economists have learned about inflation phenomena will be offered as a motivation for the focus of this pilot study. Given the scope of this writeup concepts from economic theory have been simplified, thus avoiding in-depth explanations rooted in theory and mathematical models. Notwithstanding, this brief survey's goal is to justify the relevance of the Natural Language Processing application within economics as well and to set up the problem.

One tenant of economic theory is that in the long run prices would increase after a positive demand shock. A positive demand shock is caused by an increase in consumption, investment, government spending, or net exports. Inflation (price increases) as a result of demand shock is a phenomenon that has been observed across most of the recorded economic history, this is called demand-pull inflation. Likewise, a negative supply shock would cause prices to increase (i.e. a shortage of goods caused by increase in energy prices). This later situation is called cost-push inflation.

One of the tenants of the Monetarist theory, which was established in the 1970s, is that money supply is correlated with inflation. If the money supply grows at a faster rate than the economy's ability to produce goods and services, then inflation will result.

Money supply is also related to another economic model known as "the Phillips curve." The Phillips curve is an economic model that predicts a correlation between reduction in unemployment (that is high employment) and increased rates of wage rises within an economy.

In short, this theory follows these arguments: when unemployment falls it causes workers to have more market power; this enables employees to demand higher wages. With higher wages workers have more purchasing power, which in turn increases aggregated demand. This increase in consumption causes prices to rise.

On the other hand, because prices are rising, employees find they cannot afford their lifestyles with current wages. Given that workers have greater market power when there is low unemployment -i.e. employers are competing to get people hired- workers can either demand further wage increases or find other jobs positions with higher wages.

In the meantime, employers, facing higher labor costs, need to pass these higher costs on their customers, which fuels further inflationary forces.

This situation creates a vicious cycle. Economists argue that regulating the supply of money, via higher interest rates, will help 'cool down the economy', thus stopping the inflation cycle.

According to some economist, the employees enhanced market power can also be created by cash stimulus provided by the government, such as: the American Rescue plan that was meant to provide relief to families from the economic shock caused by the COVID-19 pandemic.

During these past three years other issues have contributed to the rise in inflation: the supply and logistic bottleneck created during the pandemic, which added notoriously higher logistics costs, or the increase in energy costs as a result of the war in Ukraine caused by the Russian Federation.

Furthermore, there are other well-known issues that can contribute to and/or accelerate inflationary pressures. A few examples include: economies that cannot control the currency exchanged in their markets, such as countries that use paper money printed by other sovereign nations; or economies that are affected by other economies' inflationary pressures, given that they suffer a long-term negative trade balance.

The inflation issue this writeup will be focusing on is the "self-fulfilling prophecy" theory. This theory was popularized by sociologist Robert K. Merton. A self-fulfilling prophecy is a prediction that comes true, at least in part, as a result of a person's belief or expectation that said prediction would come true.

If inflation is expected by the public, then producers will expect higher production costs for the goods they produce; meaning, they will have to add a higher markup to their products to be able to replenish their inventories and pay for increasing labor costs. On the other hand, customers that expect inflation will accept higher prices for the same goods.

In this writeup it is argued that with the proper market conditions (i.e. asymmetry of information), this self-fulfilling-prophecy loop can be aggravated by the fact that it also creates an opportunity for producers to hike up prices, even if there is no evidence of inflation in their industry yet.

Currently policymaking organizations use surveys to track inflation expectations. However, people tend to provide inaccurate/biased information on surveys. Moreover, nowadays fewer people are willing to participate in such surveys.

A tenant of this pilot study is that inflation expectation is established by social discourse: what people are talking about. If pundits show up on cable news primetime announcing more inflation, they will be inducing the audience to expect inflation. If people in the same social field face higher food costs, they will be talking about it and this will contribute to creating an expectation of inflation in the future.

Nowadays, thanks to the internet we have an organized record of what experts said in the newspapers, what pundits said in cable news, what people discussed in public fora, and what people searched on popular search engines.

This paper argues that this organized record would be more accurate than surveys to measure inflation expectations. It will be a more empirical approach focused on what people actually do (hear, talk, is concerned about, or search), rather than on what people say.

Another hypothesis made on this brief study is that, within the scope of finance and economics, social discourse is heavily influenced by experts and pundits who spread discourse by being interviewed by the most influential and reputable news sources. In other words, it is assumed that cable news and social media discourse echo what experts said in print media, even if the content of the original speech might be taken out of context during this process.

The goal of this Natural Language Processing (NLP) project is to develop an algorithm to classify the expectation of inflation in social discourse recorded as text on the internet. Given the time limitations of this project the project, it will focus on a limited amount of news articles from reputable sources to train the NLP algorithms.

This project will be relevant to policymakers interested in measuring and operationalizing inflation expectations. Perhaps information extracted from this methodology can be used as an instrumental variable in the Federal Reserve's nowcasting methodology. Information derived from this NLP methodology can also lead to better inflation-prediction models as well as better measuring of the effect of policy interventions targeted to either ameliorate inflation, reduce inflation expectations, or curve the opportunity to create artificial price hikes.

II. Data Collection

The data was collected in two stages. First, training data and then, crawled data to be analyzed using the trained NLP model.

To pinpoint the right sources to collect news articles from, which would be fed to the NLP models during the training stage, a search was carried out to identify the most important finance and business print news outlets. The selected news sources were: The Economist, Bloomberg Businessweek, The New York Times, and The Wall Street Journal.

In the case of national newspapers, sample articles were selected from either the economics, business, or finance-and-markets sections. Care was placed in selecting news sources holding both: monetary hawkish views as well as monetary dovish views.

The sample articles were obtained using the "Business Source Complete" database available at the University of Maryland Libraries. This database did not include articles from The Economist. A different database was used to gain access to articles published by The Economist.

From a variety of pre-selected articles, 50 sample articles were selected to be read and labeled. The publication dates of the articles spanned from October 2019 through January 2023. The dates were selected to cover the timespan of the recent inflation spike, which started amidst the COVID-19 pandemic.

One of three labels were assigned to each article depending on the position of the article regarding future inflation: "Expect inflation", "Inflation will fade away", and "Neutral".

The need for a “Neutral” category was discovered while labeling the news articles. Neutral articles were either articles that wanted to act as explainers of the inflation phenomena or articles seeking to prepare the public about how to respond to different inflation scenarios.

In addition to the main body of the article, the following metadata was collected for each article: source, date, section, byline (author), number of words, inflation position (assigned label), title, and subtitle. Each article, and its metadata, was saved as a text file (.txt extension). This format was selected because it was easier to load than pdf in python. The following is an example of the structure of the text files.

```
_____________________________________________________________
| source: "The New York Times"                               |
| date: 2022-08-25                                           |
| section: "Section B; Column 0; Business/Financial Desk; Pg. 2"  |
| byline : "Ana Swanson"                                     |
| words: 488                                                 |
| inflationPosition: "Expect Inflation"                      |
|                                                            |
| title: "Consumer Demand Is Key To Rise in U.S."            |
| subtitle: ""                                               |
|                                                            |
| ###body###                                                 |
|                                                            |
| Main body of the article                                   |
|                                                            |
| ###end###                                                  |
|____________________________________________________________|
```

Keeping metadata is important because the hypothesis that either the news source or the byline (author) are better predictors of the labels than the text on the article must be rejected.

Once the news articles were saved in plain text format it was noted that words were broken during the process of changing the format from .pdf to .txt. In order to feed high quality data to the NLP models, the texts were reviewed manually to restore the words to its correct orthographic form.

Initially it was theorized that the sample articles should be loaded as python dictionaries to hold the metadata. However, in order to be sure to keep the sample articles and their labels together, a class constructor was used to create NamedTuples. For this iteration, only the body and the label were kept.

The dictionary was discarded as a data structure to hold the sample labeled articles because python dictionaries are not ordered. In this experiment, Sci-kit-learn’s command **train_test_split()** must iterate over two sequences, one holding the sample articles and another holding the labels. If we let the command iterate over two unordered data structures (one iteration for the body of the articles and a second iteration for the labels), we have no guarantee of sample articles and their corresponding labels to end up in parallel indexes once the train-test splits algorithm finishes. It is because of this reason that we stored the sample articles and labels in a list of NamedTuples. The list is an ordered structure and

NamedTuple is an immutable type, this structure guarantees sample articles and their corresponding labels to remain in the same indexes, both prior and after the train-test split algorithm is run. It also guarantees that the tuple cannot be accidentally reassigned.

In order to validate the hypothesis that cable news and social media (the super spreaders of discourse in society) are mere participants in a "transmission chain experiment game" and the hypothesis that "ground zero" of social discourse within the scope of finance and economics is specialized printed media, a larger dataset of cable news, tabloid news, newsletters, and social media posts was needed.

The larger dataset was obtained from Common Crawl. In order to setup the script to download the WARC file from Common Crawl and then setup the script to extract the articles in the WARC file, code from the NewsPlease module, created by Felix Hamborg (https://github.com/fhamborg/news-please), was adapted and refactored. Among the contributors to this module is Sebastian Nagel who is part of the Common Crawl foundation.

From the refactored code from the NewsPlease module the **commoncrawl_crawler.crawl_from_commoncrawl()** command was used with the following arguments:

```
############ YOUR CONFIG ############
# download dir for warc files
my_local_download_dir_warc =
'C:/Users/rzamb/Desktop/Desktop/UMD/641_Natural_Language_Processing/finalProject/cc_download_
warc/'
# download dir for articles
my_local_download_dir_article =
'C:/Users/rzamb/Desktop/Desktop/UMD/641_Natural_Language_Processing/finalProject/cc_download_
articles/'
# hosts (if None or empty list, any host is OK)
my_filter_valid_hosts =
['cnn.com','nytimes.com','msn.com','foxnews.com','bbc.com','finance.yahoo.com','washingtonpost.com',
'usatoday.com','cnbc.com','theguardian.com','wsj.com','ft.com','reddit.com','quora.com','economist.com
','bloomberg.com','latimes.com','bostonglobe.com','miamiherald.com','nypost.com','newyorker.com','na
tionalreview.com','breitbart.com','reuters.com','pbs.org','npr.org','apnews.com','substack.com','medium.
com','vox.com', 'aeon.com','forbes.com','wired.com','time.com','worldbank.org','theatlantic.com',
'huffingtonpost.com','cbsnews.com','fortune.com','nbcnews.com','politico.com','huffpost.com',
'nymag.com','dailymotion.com','foxbusiness.com','mashable.com','chicagotribune.com','cbs.com','aljaze
era.com','hbr.org','brookings.edu','fastcompany.com','entrepreneur.com','theonion.com','foreignpolicy.c
om','c-span.org','slate.com','proquest.com','democracynow.org','tampabay.com','startribune.com',
'newsday.com','newsmax.com','seattletimes.com','dallasnews.com','denverpost.com','sfchronicle.com','c
hicago.suntimes.com','inquirer.com','azcentral.com','babylonbee.com','bbc.co.uk','people.com']
# example: ['elrancaguino.cl']
# start date (if None, any date is OK as start date), as datetime
my_filter_start_date = datetime.datetime(2019, 10, 1)  # datetime.datetime(2016, 1, 1) ## Default =
None
```

```
# end date (if None, any date is OK as end date), as datetime
my_filter_end_date = datetime.datetime(2023, 2, 28)  # datetime.datetime(2016, 12, 31) ## Default = None
# Only .warc files published within [my_warc_files_start_date, my_warc_files_end_date) will be downloaded.
# Note that the date a warc file has been published does not imply it contains only news
# articles from that date. Instead, you must assume that the warc file can contain articles
# from ANY time before the warc file was published, e.g., a warc file published in August 2020
# may contain news articles from December 2016.
my_warc_files_start_date = datetime.datetime(2022, 10, 1) # example: datetime.datetime(2020, 3, 1) ## Default = None
my_warc_files_end_date = datetime.datetime(2023, 2, 28) # example: datetime.datetime(2020, 3, 2) ## Default = None
# if date filtering is strict and news-please could not detect the date of an article, the article will be discarded
my_filter_strict_date = True
# if True, the script checks whether a file has been downloaded already and uses that file instead of downloading
# again. Note that there is no check whether the file has been downloaded completely or is valid!
my_reuse_previously_downloaded_files = True
# continue after error
my_continue_after_error = True
# show the progress of downloading the WARC files
my_show_download_progress = False
# log_level
my_log_level = logging.INFO
# json export style
my_json_export_style = 1  # 0 (minimize), 1 (pretty)
# number of extraction processes
my_number_of_extraction_processes = 1
# if True, the WARC file will be deleted after all articles have been extracted from it
my_delete_warc_after_extraction = True
# if True, will continue extraction from the latest fully downloaded but not fully extracted WARC files and then crawling new WARC files. This assumes that the filter criteria have not been changed since the previous run!
my_continue_process = True
# if True, will crawl and extract main image of each article. Note that the WARC files
# do not contain any images, so that news-please will crawl the current image from
# the articles online webpage, if this option is enabled.
my_fetch_images = False
# if True, just list the WARC files to be processed, but do not actually download and process them
my_dry_run=False
############ END YOUR CONFIG #########
```

The news articles extracted from the WARC file were in .json format. Each .json file would have the following metadata:

authors
date_download
date_modify
date_publish
description
filename
image_url
language
localpath
maintext
source_domain
title
title_page
title_rss
url

When the filter to limit articles to only those written in English (data['language'] = 'en') was applied, a total of 13,096 were extracted.

It was noticed that text from some of the extracted articles were showing a message in the 'maintext' field instead of the actual original text of the article. The text was a message stating that the article was subject to copyright and thereby it was only available to subscribers. These articles were filtered out when selecting a subset of the crawled articles. This subset would be the crawled articles that had the word 'inflation' in the body of the news article.

The WARC file was published in November 2022. The span of the dates for the crawled articles overlapped the latest episode of inflation spike in the United States. Namely, from January 2020 through October 2020.

The final note regarding data preprocessing is that all articles, including sample articles and the larger set of news articles had to be loaded using the utf8 encoding. This was necessary to avoid corruption of the files and generating the so-called 'gremlins' (text composed of unreadable/meaningless symbols).

Annotator agreement metrics are not included in this pilot experiment given that the sole annotator labeling the inflation position of the sample articles was the author of this term paper. This annotation task would need to employ economists. This limitation made it hard to find and recruit other annotators.

III. Methodology

3.1. Train-Test Split

The first step of the methodology was splitting the labeled data into a training set and a test set.

The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model. In other words, to test how well the models generalize to unseen data. When enough labeled data is available a validation set is included.

The training set is used to fit the models, the validation set is used to estimate prediction error for model selection, and the test set is used for assessment of the generalization error of the final chosen model.

Ideally the test set should be used only after the data analysis has been done. If the machine learning algorithm "sees" the testing set during the training stage, there is a higher risk of overfitting the model to the test data, thus producing an artificially low error. The goal is to evaluate how well the machine learning algorithm generalizes to "unseen" data.

The size of the test set in this experiment was established to 20% of the labeled data. Given that there was an unbalanced set of labeled examples, a larger test set was needed to capture instances of the minority classes. We could have used stratified train-test split, but we used the simplest option available. Most of the sample articles were labeled "Expect inflation" (the majority class), which was not surprising since the United States is going through an inflationary spiral.

3.2. Articles Encoding

Different encoding algorithms were used to build numeric representations of text sequences. Machine learning algorithms can only process numbers, encoding algorithms transform text sequences (words, phrases, documents) into numeric inputs suitable for machine learning models. For this experiment the following encoding algorithms were used: bag of words (BOW), Term Frequency - Inverse Document Frequency (Tf-idf), doc2vec, and GloVe (global vectors for word representation).

3.2.1. Bag of Words (BOW)

Bag of words is an encoding algorithm that represents documents by keeping the frequency of all the words occurring in the document. The order of the words is ignored. Each word is considered a feature and it will vary from document to document by the number of occurrences in each one.

The vocabulary of the encoding will be composed of all words occurring in all documents in the training set. If a given document has no occurrence of a particular word, the word's count will be equal to zero. In the test set, if a document has a word that did not occur in the training set, this word is ignored; this is because the word is outside of the vocabulary.

This representation leads to sparse vectors. These are long vectors with many features equal to zero.

To encode documents using BOW scikit-learn's **feature_extraction.text.CountVectorizer()** was used. This command accepts the following keyword arguments: stop_words and ngram_range.

Stop words are a set of commonly used words in a given language. The use of stop words eliminates unimportant words/features, words that convey little information from the document. In this experiment the Mallet set of stop words for English language was used.

N-grams are continuous sequences of n words or symbols in a document. The n-gram range in the **CountVectorizer()** command allows the function's client to specify the range of n-grams to be included in the vocabulary. In this pilot study the specification used bi-grams, tri-grams, and 4-grams.

3.2.2. Term Frequency - Inverse Document Frequency (Tf-idf),

Tf-idf is a numerical statistic, used for information retrieval, that is intended to reflect how important a word is to a document in a collection or corpus. This is an encoding technique that uses term weighting. The weights are used to assign higher importance to words that occur only in a few documents. Terms that are limited to a few documents are useful for discriminating the documents in a given corpus.

Tf-idf weighting is the product of two terms: term frequency and document frequency. Term frequency is the frequency of word "t" in the document "d", and it can be computed as $tf_{t,d} = \log_{10}(\text{count}(t,d)+1)$. The document frequency df_t of a word "t" is the number of documents, from a given corpus, word "t" occurs in. Document frequency is different from collection frequency, the latter is the total number of times the word appears in the whole collection. Idf is calculated as $\log_{10}(N/df_t)$, where N is the total number of documents in the collection.

The weighted value $w_{t,d}$ for word "t" in document d combines term frequency with inverse document frequency: $w_{t,d} = tf_{t,d} * idf_t$.

The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

A high weight in tf–idf is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms. Since the ratio inside the idf's log function is always greater than or equal to 1, the value of idf (and tf–idf) is greater than or equal to 0. As a term appears in more documents, the ratio inside the logarithm approaches 1, bringing the idf and tf–idf closer to 0.

To encode documents using BOW scikit-learn's **feature_extraction.text.TfidfVectorizer()** was used. This command accepts the following keyword arguments: stop_words and ngram_range.

3.2.3. GloVe Embeddings

Both the BOW as well as the tf-idf encoding represent documents as a sparse long vector with dimensions equal to the words in the vocabulary. A document encoded using either of the aforementioned techniques can have 50,000 features or more. In order to get better results, short dense vectors are preferred.

Embeddings represent words as short and dense vectors. There are two well-known methodologies to represent words as embeddings: word2vec and GloVe. Pre-trained embeddings using these two methodologies are available online.

In this experiment both embeddings were tested. It was decided to use GloVe embeddings. While testing GloVe embeddings, it was found that words embeddings similar to the word embedding for "inflation" were words that an economist would think of as related to inflation. These similar words embeddings were found using cosine similarity in the vector space of the embeddings. See figure 1 and 2 to inspect the results obtained.

Figure 1 – Embeddings similar to inflation using word2vec

```
sims = model.most_similar('inflation', topn=10)
print(sims)
```

```
[('Inflation', 0.8240750432014465), ('inflationary_pressures', 0.8009414076805115), ('inflationary', 0.7951276302337646),
('inflationary_pressure', 0.7671657800674438), ('inflation_pressures', 0.7616896629333496), ('inflationary_expectations', 0.
7339735031127 93), ('CPI', 0.6960082650184631), ('deflation', 0.6912639737129211), ('disinflation', 0.6745432615280151), ('in
terest_rates', 0.6639971733093262)]
```

Figure 2 – Embeddings similar to inflation using GloVe

```
sims = model.most_similar('inflation', topn=10)
print(sims)
```

```
[('rate', 0.7825247645378113), ('rates', 0.7814398407936096), ('unemployment', 0.7501060366630554), ('inflationary', 0.74411
07630729675), ('growth', 0.7435855269432068), ('deflation', 0.7362003326416016), ('rising', 0.732311487197876), ('rise', 0.7
258995771408081), ('slowing', 0.715856671333313), ('prices', 0.7068049311637878)]
```

Genism module is an open-source Python library for representing documents as semantic vectors (https://radimrehurek.com/gensim/index.html). Gensim was used to load the pretrained vectors. The pretrained GloVe embeddings were downloaded from: https://nlp.stanford.edu/projects/glove/.

GloVe embeddings were used to build the embedding matrix used in neural network architecture for classification. When building the embedding matrix, words that were not in the vocabulary of the pre-trained embeddings were set up as a vector filled with zeros. Out of 4,984 words 498 were not in the pre-trained GloVe embeddings.

3.2.4. Doc2Vec

Doc2vec is a generalization of the wrod2vec model. In creates short and dense vectors to represent documents as embeddings. Once documents in a training set are encoded in embedding vectors using doc2vec, cosine similarity can be used in a corpus of new documents to find which document in the

training set a given new document is closer to in the embedding vector space. The goal is to identify which document in the training set a given new document is more similar to. Then the new document is assigned the same label the most similar document has.

Gensim's package models.doc2vec was used to encode the training documents as well as to infer the embedding vectors of new documents.

Figure 3 showcases the inferred vector embedding for a one-phrase document.

Figure 3 – Inferred embedding vector for a document containing only one phrase

```
# Let's check the document embedings
vector = model.infer_vector("The unemployment rate is too low and this fuels inflation".split())
print(vector)
```

```
[-0.21959691 -0.16523121  0.03144318  0.31085932  0.1098144   0.04731079
  0.15995571 -0.00967111 -0.21459682  0.30741775 -0.20229432  0.16568355
  0.10838725  0.08592039 -0.16812086 -0.2548607   0.11954755  0.39605606
 -0.02387948  0.10477146  0.1258252   0.13008118 -0.22675449  0.10589869
 -0.0702756  -0.23844709 -0.3180193  -0.21744446 -0.16108482 -0.43238148
  0.12040122  0.11571792  0.11666974  0.09695289 -0.04072885  0.20876156
  0.02342697 -0.18610595  0.20022662  0.06176469  0.08886606 -0.15506977
  0.08286011 -0.09290649  0.16938923  0.21108295  0.02639672 -0.2749316
  0.11128721 -0.25525537]
```

3.3. Labels Encoding

As it was mentioned earlier, machine learning algorithms can only process numbers. Hence, there is a need to encode the labels as well.

For the neural network models, the algorithms expect the labels to be encoded as dummy variables. That is, a value equal to 1 for the category the sample article belongs to and 0 for the other categories. For the non-neural-network models the algorithm expects just one variable with discrete levels, one for each category.

For the discrete levels encoding, sci-kit learn was used. Scikit-learn has a preprocessing module which offers the LabelEncoder() command. This command encoded each category as follows:

'Expect inflation' = 0
'Inflation will fade away' = 1
'Neutral' = 2

To encode the levels as dummy variables, scikit-learn's LabelEncoder() command was used first and then the to_categorical() command from Kera's util module. In this encoding the output would be a matrix. Each row would correspond to a single labeled observation. The first column of the matrix would correspond to 'Expect inflation', the second column to 'Inflation will fade away' and the third column to 'Neutral'. On each row a value of one will appear in the column corresponding the sample's label and zero for the rest.

3.4. Classification Models

The following is a brief survey of the machine learning models used in this experiment.

3.4.1. Naïve Bayes

The Naïve Bayes classifier is a supervised machine learning algorithm, which is used for classification tasks. Naïve Bayes belong to the family of simple "probabilistic classifiers" and it is based on applying Bayes' theorem to the data. The probability of each class “c” conditioned on a given document “d” is computed from the probability of the document “d” conditioned on each class “c” (likelihood) times the probability of each class “c” (prior).

$P(c|d) = P(d|c)*P(c)$

The denominator P(d) is the same for each class, so it cancels out when computing the probability of each class. Given that each document can be represented as a set of features, the prior $P(d|c)$ becomes $P(f_1, f_2 \dots f_n|c)$.

A naïve Bayes classifier is called naïve because it makes a simplifying assumption about how the features, words in this case, interact with each other. The simplification, commonly called the “naïve Bayes assumption”, states that there is conditional independence between the features’ probabilities given each class $c \in C$ (C is defined as the set of classes).

Then, $P(f_1, f_2 \dots f_n|c) = P(f_1|c)*P(f_2|c)*\dots*P(f_n|c)$

The final equation for the class chosen by a naive Bayes classifier is:

$C_{NB} = \text{argmax}_{c \in C} P(c)*[P(f_1|c)*P(f_2|c)*\dots*P(f_n|c)]$

Scikit-learn’s **naive_bayes.MultinomialNB()** was used to setup the model.

The naïve Bayes model with the BOW encoding was used as the ***baseline*** model in this experiment.

3.4.2. Support Vector Machines (SVM)

Support vector machines is a supervised learning model. It can be used for both: classification and linear regression. It can perform non-linear classification by using the “kernel trick”.

The support vector machines algorithm finds the hyperplane that separates classes with the maximal margin.

Scikit-learn’s **svm.SVC()** was used to setup the model. The following hyperparameters were used: C=1.0, kernel='linear', degree=3, gamma='auto'.

3.4.3. Cosine Similarity

To measure similarity between two target words v and w (or two documents d1 and d2) encoded as vector embeddings, cosine similarity is a popular metric. It measures the angle between two embedding vectors that represent words (or documents).

The cosine is based on the dot product of two vector embeddings: dot_product(v,w).

The dot product will tend to be high just when the two vectors have large values in the same dimensions. Alternatively, vectors that have zeros in different dimensions—orthogonal vectors—will have a dot product of 0, representing their strong dissimilarity.

Since the dot product favors long vectors, a normalized dot product is used. To normalize the dot product, the dot product is divided by the length of each of the two vectors. This normalized dot product turns out to be the same as the cosine of the angle between the two vectors.

$$\frac{v \cdot w}{|v||w|} = cos\theta$$

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for orthogonal vectors, to -1 for vectors pointing in opposite directions. But since raw frequency values are non-negative, the cosine for these vectors ranges from 0–1.

Gensim's models.doc2vec module was used to find cosine similarity between documents.

3.4.4. Long Short-Term Memory Recurrent Neural Network (LSTM)

Long short-term memory (LSTM) is an extension of recurrent neural networks. It was developed by Hochreiter, S. and J. Schmidhuber in 1997.

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes can create a cycle, allowing output from some nodes to affect subsequent input to the same nodes. This allows it to exhibit temporal dynamic behavior. Because of this RNNs can be used to analyze temporal phenomena, such as: economic time series, weather observations, or language.

Recurrent neural networks can process sequences of data such as texts. However, these networks are difficult to train. In particular, it is quite difficult to train RNNs for tasks that require a network to make use of information distant from the current point of processing. This is because despite having access to the entire preceding sequence, the information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence and recent decisions. Yet distant information is critical to many language applications.

LSTMs divide the context management problem into two subproblems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. It accomplishes this by adding an explicit context layer and through the use of specialized neural units that

make use of gates to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, previous hidden layer, and previous context layers.

The gates in an LSTM share a common design pattern; each consists of a feedforward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated.

A common LSTM unit is composed of a forget gate, an add gate, and an output gate. In brief, the purpose of the forget gate is to delete information from the context that is no longer needed. The add gate selects the information to add to the current context. The output gate is used to decide what information is required for the current hidden state.

The architecture of this model used Keras' LSTM() layer in combination with two Kera's Dense() layers. See the model's printout below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 300)               0
_________________________________________________________________
embedding (Embedding)        (None, 300, 100)          469300
_________________________________________________________________
lstm (LSTM)                  (None, 128)               117248
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 128)               16512
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 387
=================================================================
Total params: 603,447
Trainable params: 603,447
Non-trainable params: 0
_________________________________________________________________
```

A bidirectional version of the LSTM model was also tested. This is the model's printout:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         (None, 150)               0
_________________________________________________________________
embedding_2 (Embedding)      (None, 150, 100)          469300
_________________________________________________________________
bidirectional (Bidirectional (None, 256)               234496
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_10 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_11 (Dense)             (None, 3)                 387
=================================================================
Total params: 737,079
Trainable params: 737,079
Non-trainable params: 0
_________________________________________________________________
```

3.4.5. Gated Recurrent Unit Recurrent Neural Network (GRU)

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014 by Kyunghyun Cho et al. The GRU is like a long short-term memory (LSTM) with a forget gate, but has fewer parameters than LSTM, as it lacks an output gate.

The architecture of this model used Keras' GRU() layer in combination with two Kera's Dense() layers. See the model's printout below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 150)               0
_________________________________________________________________
embedding_2 (Embedding)      (None, 150, 100)          469300
_________________________________________________________________
gru (GRU)                    (None, 128)               87936
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_8 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_9 (Dense)              (None, 3)                 387
=================================================================
Total params: 574,135
Trainable params: 574,135
Non-trainable params: 0
_________________________________________________________________
```

A bidirectional version of the GRU model was also tested. This is the model's printout:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_10 (InputLayer)        (None, 150)               0
_________________________________________________________________
embedding_4 (Embedding)      (None, 150, 100)          469300
_________________________________________________________________
bidirectional_2 (Bidirection (None, 256)               175872
_________________________________________________________________
dropout_9 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_18 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_19 (Dense)             (None, 3)                 387
=================================================================
Total params: 678,455
Trainable params: 678,455
Non-trainable params: 0
_________________________________________________________________
```

For each of the neural network architectures stop words were used to filter out common words from the texts. Furthermore, from the remaining words, only the top 5,000 words were used. To select the most relevant 5,00 words **keras.preprocessing.text.Tokenizer()** was used.

3.5. Model Assessment

3.5.1. Accuracy

Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions the model got right. For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

3.5.2. Precision

Precision is a metric that pinpoints the proportion of positive identifications that were actually correct. It is given as:

$$Precision = \frac{TP}{TP + FP}$$

3.5.3. Recall

The recall metric provides information about the proportion of actual positives that was identified correctly. It is defined as:

$$Recall = \frac{TP}{TP + FN}$$

In general, accuracy is a better metric to evaluate models with balanced data. Precision and recall are better metrics for models with imbalanced training data. In this experiment the data was imbalanced:

	Number of Labeled Samples
Expect inflation	25
Inflation will fade away	15
Neutral	5

To evaluate a model both precision and recall are needed. However, when attempting to improve a model it has to be borne in mind that improving precision typically reduces recall. For example, when fine-tunning a model to decrease the number of false positives (improving precision), false negatives increase as a consequence (thus recall decreases).

In this pilot study, when deciding which model has the best performance, accuracy will be a salient metric. Furthermore, in this experiment it was assumed that the cost of false negatives is higher. That is, allowing news articles that foster a higher expectation of inflation will create an undesirable bias that would have a negative impact in the policymaker's decision process. Thereby, on the final assessment models with higher recall will be favored.

IV. Results and Discussion

Each model architecture was run twice with the same hyperparameters. This was as a result of seeking to include per-class precision and recall.

On the first run the train-test split was run several times in order to guarantee that samples from all categories were included in the test set. On the second run the test set did not include samples from the minority class: 'Neutral'.

This fact changed the accuracy of all models. In particular, it was observed that the accuracy of the models was better when predicting the labels of sample articles in the test set without samples from the minority class.

Another fact that was observed when comparing these two runs was that the predicted labels were overfitted more frequently in the runs that included the minority class in the test set. In these cases, the prediction defaulted to the majority class. It has to be noted that in the first run a handful of the models

performed worse than just predicting the labels by defaulting all predictions to the majority class. Figure 4 showcases the predictions of an overfitted model.

Figure 4 – Overfitted model

```
print(predictions_SVM)
print(y_test_encoded)
```

```
[2 0 0 0 0 0 0 0 0 0]
[0 1 0 2 0 0 1 2 0 0]
```

Two conclusions comes to mind from these observations. First, although it might be difficult to collect a balanced set of labeled examples (because of the seasonality and the political coverage of inflation phenomena), more samples from the minority classes need to be added to the training set.

Second, when selecting the model, K-fold validation should be used in order to have better estimates of the models' performance. K-fold validation was not used in this pilot study to save time in the training stage.

The following tables summarize the results obtained for each one of the models. The first table showcases results from the first run and the second table results from the second run. Rows highlighted in green were considered the best performing models.

Table 1 – Performance of different models and NN architectures. First run.

Methodology/Architecture	Encoding	loss	accuracy	Overfitted	Max. Length	Learning Rate	Epochs
bidirect-GRU	GloVe	1.172127366	0.40	No	150	0.00001	100
bidirect-LSTM	GloVe	1.104177475	0.40	No	150	0.00001	100
GRU	GloVe	1.081696749	0.40	No	150	0.000001	150
GRU	GloVe	1.090383768	0.40	Yes	300	0.00001	100
GRU	GloVe	1.069840789	0.40	No	150	0.00001	100
LSTM	GloVe	8.224319458	0.40	Variation but with 100% certainty	150	0.0005	100
LSTM	GloVe	nan	-	Not Applicable	500	0.0005	100
LSTM	GloVe	nan	0.40	Not Applicable	300	0.0001	150
LSTM	GloVe	1.206672072	0.30	No	300	0.00001	100
LSTM	GloVe	9.67085743	0.40	Variation but with 100% certainty	300	0.0001	100
Cosine Similarity	doc2vec	-	0.50	No	-	-	-
SVM	tf-idf	-	0.50	Yes	-	-	-
Naïve Bayes	tf-idf	-	0.50	Yes	-	-	-
SVM	BOW	-	0.50	Yes	-	-	-
Naïve Bayes (Baseline)	BOW	-	0.20	No	-	-	-

Table 2 – Performance of different models and NN architectures. Second run.

				'Expect inflation' category			Hyperparameters		
Methodology/Architecture	Encoding	loss	accuracy	precision	recall	Overfitted	Max. Length	Learning Rate	Epochs
bidirect-GRU	GloVe	0.7772	0.40	0.40	1.00	No	150	0.00001	100
bidirect-LSTM	GloVe	0.8564	0.40	0.40	1.00	No	150	0.00001	100
GRU	GloVe	1.0381	0.50	0.43	0.75	No	150	0.000001	150
GRU	GloVe	1.0982	0.40	0.40	1.00	Yes	500	0.000001	150
GRU	GloVe	1.0539	0.40	0.40	1.00	Yes	300	0.00001	100
GRU	GloVe	0.9588	0.40	0.40	1.00	No	150	0.00001	100
LSTM	GloVe	0.5981	0.40	0.38	0.75	No	150	0.00005	100
LSTM	GloVe	8.059	0.30	0.33	0.75	No	300	0.0001	150
LSTM	GloVe	0.9544	0.60	0.50	1.00	No	300	0.00001	100
LSTM	GloVe	9.2679	0.70	0.67	0.50	No	300	0.0001	100
Cosine Similarity	doc2vec		0.70	0.75	0.86	No	-	-	-
SVM	tf-idf	-	0.70	0.70	1.00	Yes	-	-	-
Naïve Bayes	tf-idf	-	0.70	0.70	1.00	Yes	-	-	-
SVM	BOW	-	0.40	0.57	0.57	No	-	-	-
Naïve Bayes (Baseline)	BOW	-	0.70	0.75	0.86	No	-	-	-

4.1. Baseline

The baseline model used the simplest encoding, that is BOW, combined with the Naïve Bayes multinomial classification model.

On the first run, with an accuracy of 0.2, it showed poor performance. However, when it did not have to predict the minority category its accuracy was improved to 0.7. The recall per class for the first run was not calculated, in the second run it was shown that the recall for the majority class was 0.86. That is, out of all the sample articles labeled as 'Expect inflation', the model predicted the label correctly for 86% of labeled articles.

In the second run the baseline performance was strong.

4.2. Support Vector Machines

Across all the runs and encodings, the SVM algorithm was overfitted. For an expansion of this study the SVM might be tried again with different hyperparameters.

4.3. Cosine Similarity

The cosine similarity algorithm encoded with doc2vec performed well on both runs. This is definitely a model to try out in an expansion of this pilot study.

An advantage of this methodology is that it takes into account the complete text.

As it will be seen below, neural network architectures will use only a limited number of words from the articles. Different authors will have different writing styles, meaning the main idea of a text can be delivered at different locations within the text. Articles written by authors who prefer to write main ideas at the end of the text (because arguments are presented first or because the narrative style seeks to

surprise the reader with a spin), will probably be mislabeled consistently by neural network architectures given that these models would be using only the first words of the article.

The latter concern would not affect short texts (like tweets) or systematically structured documents such as hard news articles that open the text with a lead. The issue might affect opinion articles or social media posts more critically.

4.4. Neural Network Architectures

Two recurrent neural network architectures were explored: LSTM and GRU.

On the first run GRU architectures consistently performed better over LSTM architectures. On the second run LSTM architectures improved their performance. In neither case the neural network architectures performed better than cosine similarity.

Bidirectional GRU and LSTM architectures were tested. However, they did not outperform the simple LSTM and GRU architectures.

From the limited hyperparameter search it was observed that reducing the learning rate as well as increasing the epochs had a positive effect in the model's performance. Surprisingly, increasing the size of words fed to the algorithm had a negative effect in the performance of the models.

Although neural network architectures did not do better than cosine similarity, they should not be discarded for an expansion of this pilot study. Fifty labeled samples are far from enough to train an effective neural network model.

4.5. Classifying Articles from Common Crawl

For this pilot study only articles from cable news websites, national newspapers, magazines, and tabloids were analyzed. Newsletters and social media posts were discarded.

Articles were classified using the cosine similarity methodology and using the GRU neural network architecture.

Initially, it was observed that the classified articles were unrelated to inflation. For example, a crawled article showing a high cosine similarity to articles labeled as 'Expect inflation' was titled "Cecily Strong leaves 'SNL'? Not just yet".

The methodology was forced to classify any article, of any topic, into tight categories related only to the topic of inflation. An article about 'Mediterranean food' would have to be fitted into either 'Expect inflation', 'Inflation will fade away', or 'Neutral' (recall the neutral category was still related to inflation). Thereby, articles were filtered out using the following criteria: select only articles that have the word 'inflation' in the main body of the article. Out of the 13,096 crawled articles only 517 mentioned 'inflation'.

In a further expansion of this pilot study, it would be required to include a category labeled ‘unrelated’ and labeled examples of ‘unrelated’ texts would need to be provided. Otherwise, articles with topics unrelated to ‘inflation’ will be forced into one of the proposed categories, thereby causing noise in the output/prediction of the classification algorithm.

Once the filter was applied, it was observed that there was little agreement/overlap between the predictions made by the cosine similarity methodology and the GRU model.

Upon visual inspection of the text of a handful of the crawled articles it was found that the cosine similarity algorithm was more accurate than the GRU architecture when assigning labels to the articles. It was also noted that even after filtering out crawled articles using the ‘include-the-word-inflation’ criteria there were still articles that should have been labeled as ‘Unrelated’.

V. Future Work and Limitations

The first limitation found was time. With more time this pilot study can be expanded to include:

- K-fold validation to have better estimates of the models’ performance
- Further fine-tunning of hyper parameters. This is especially necessary for the SVM algorithm as well as for the neural network architectures
- Testing of a transformed model

The most important limitation of this exploratory work was the lack of abundant labeled data. Further experiments must include:

- Multiple annotators with formal training in economics
- More labeled examples for training
- A validation set

In further experiments, it will be important to introduce an ‘unrelated’ label and provide labeled examples to the training set.

In any continuation of this pilot study the hypotheses that either of the data points recorded in the metadata is a better predictor to the labels than the body of the sample articles must be evaluated. Each one of these hypotheses should be contrasted and compared with the performance of the classification models.

Finally, further research should check that crawled data does not contain messages in lieu of the original body of the news articles.

VI. Conclusion

In this pilot study it was found that a machine learning algorithm can be used to classify news articles by their position regarding inflation expectations. With as little as 50 sample articles the cosine similarity methodology combined with doc2vec encoding was able to generalize well on unseen news articles.

With the inclusion of an 'Unrelated' label and its corresponding sample articles, as well as the increasing the number of labeled articles in the training set, it is expected that both the cosine similarity methodology and the neural network architectures will perform better.

VII. References

- Daniel Jurafsky and James H. Martin, "Daniel Jurafsky and James H. Martin" 3rd. Edition (Draft).
- Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie, "The Elements of Statistical Learning" 2nd Edition (2009)