Reproducing "Mining Causal Topics in Text Data: Iterative Topic Modeling with Time Series Feedback"

Faith Chung
CS 410
Fall 2020

1. Introduction

For this course project, the Causal Topic Mining paper "Mining Causal Topics in Text Data: Iterative Topic Modeling with Time Series Feedback" by Hyun Duk Kim, Malu Castellanos, Meichun Hsu, ChengXiang Zhai, Thomas Rietz, and Daniel Diermeier was reproduced. The goal of this project is to implement the iterative topic model with time series feedback discussed in the paper to discover causal topics. The data set for this paper are NYT articles from 5/2000-10/2000 and the IEM 2000 Presidential Winner Takes All Market time series data. This was an individual project.

This paper will discuss the implementation of the causal topic mining paper, test results, and possible reasons for deviations from the paper's test results. Implementation details include preprocessing techniques for the NYT and IEM data, building the LDA model for topic mining, implementation for topic and word level causality analysis, and prior generation. Test results cover Average Causality Confidence and Average Purity calculations for situations where the strength of the prior (u) = [10,50,100,500,1000] and number of topics (tn) = [10,20,30,40], with 5 iterations for each test case. Instructions on how to run this code and install all dependencies are included in the Final Project Presentation, which is in the same directory as this project report.

This is the youtube link to the demo: https://youtu.be/K2cQH4pPyBk . You can also find the zipped video in the same directory as the report.

2. Code Overview

2.1. Dependencies

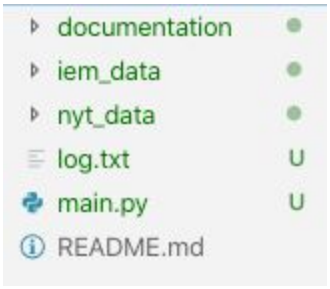
For the implementation, the code was programmed using python. The packages/libraries include:

- Used for general text io:
    - os, sys
- Used for preprocessing:
    - xml.etree
    - stop_words
    - Nltk
- Used for LDA model:
    - gensim
- Used for topic/word causality analysis:
    - statsmodels (for Granger)
    - scipy (for pearson)
- Used for miscellaneous computations:
    - numpy
    - pandas
    - collections

- datetime
- math

2.2 Directory Layout

The structure of the git repository is as follows:



The first folder, “documentation”, contains the reports and presentations for this project. The second folder “iem_data” contains “norm_prices.txt” which is a list of normalized prices used for the stock market time series data. The last folder is “nyt_data”, which contains folders for 05/2020-10/2020 containing xml files for all articles from the NYT corpus. “nyt_data” also contains text files titled “m_d_y.txt”, where each line represents a document with all paragraphs containing either candidate’s names. These txt files are generated from the preprocessing step in my code, “main.py”.

For the files, “log.txt” is generated from running the code. For each iteration of the topic model, “main.py” writes the “u” and “tn” parameters, average causality confidence, average purity, and top words from the top topics to “log.txt”. “main.py” is the only code used to run the entire iterative topic model with time series feedback. The README.md was automatically generated from github, which just provides a brief overview of the repository.

2.3 Code Execution

Please follow these steps for code execution:

1. pip install any packages you do not have in section 2.1
2. Execute: **python main.py**

Note: python 3 is recommended for running main.py

2.4 Code Overview

When main.py is executed, the main function is called. The main function executes in this order:

1. Preprocess IEM and NYT data (lines 442-452)
2. Perform strength of prior test, for each u = [10,50,100,500,1000]: (lines 462-502)
    a. Perform 5 iterations of topic model with time series feedback. For each iteration:
        i. Generate LDA (lines 473)
        ii. Generate Top Topics with LDA (lines 476-479)
        iii. Generate Top Words from Top Topics (lines 482-485)

iv. Generate Prior from Top Words (lines 488-493)
v. Update prior in LDA (line 494)
vi. Write average causality confidence and purity to log and stdout (line 496-503)

3. Perform number of topics test, for each tn = [10,20,30,40]
    a. Perform 5 iterations of topic model with time series feedback. For each iteration:
        i. Generate LDA (lines 519)
        ii. Generate Top Topics with LDA (lines 522-525)
        iii. Generate Top Words from Top Topics (lines 528-531)
        iv. Generate Prior from Top Words (lines 534-539)
        v. Update prior in LDA (line 494)
        vi. Write average causality confidence and purity to log and stdout (lines 496-503)

The code is organized so that the procedures and data for each step is encapsulated in an object. If a step is dependent on information from a previous step, that step's object is initialized with data from the previous steps. The following sections will discuss each step in more detail.

2.4.1 Preprocess IEM and NYT data

Preprocessing the IEM and NYT data is done by creating a PreProcessing object. No arguments are taken to initialize this object. The PreProcessing class contains these variables and procedures:

PreProcessing Class

Class Variable	Description
`iem_data`	List of iem data in date and normalized price pairs: *[[date1, normalized price1],[date2, normalized price 2], etc...]*
`nyt_data`	List of txt files, where each txt is 1 day's worth of preprocessed documents. *[path_to_txt1, path_to_txt2, etc…]*
`dictionary`	Dictionary that maps a unique int to each unique token/word in all documents
`doc_array`	List of tokenized documents, which is used to create the dictionary
`date_array`	List of dates, where each date at each index "i" corresponds to each doc at index "i" in doc_array

Class Procedure	Input	Output	Description
`process_iem`	None	None	Generates `iem_data` by iterating through all

			iem_data in cwd/iem_data/norm_prices.txt and only taking the dem prices, which is Gore Price / (Gore Price + Bush Price)
process_nyt	None	None	Generates nyt_data. If there are no preexisting preprocessed "txt" files (such as 05_01_00.txt) in nyt_data, this procedure goes through all xml files in cwd/nyt_data, and forms txt files for each day, where each line is each document, containing paragraphs that mention either presidential candidate's name.
set_dictionary	None	None	Takes all preprocessed files from nyt_data, tokenizes them, and creates a dictionary from all the tokens from all docs across all days.
get_iem	None	iem_data	Returns iem_data
get_nyt	None	nyt_data	Returns nyt_data
get_data	None	dictionary	Returns dictionary
get_dates	None	doc_array	Returns doc_array
get_dict	None	date_array	Returns date_array

To preprocess the data, process_iem() and process_nyt() are called first after initializing the object, and then set_dictionary() to create the dictionary. In process_nyt(), xml.etree.ElementTree was used to retrieve all text between <p> tags, and those strings were used as text data for every document.

2.4.2 Generate LDA

Once the data is preprocessed and we have a dictionary, list of tokenized documents, as well as a timestamp for all the tokenized documents, we are ready to generate the LDA model.

LDA Class

Class Variable	Description
doc_array	Array of tokenized documents, from PreProcessing
dict	Dictionary, from PreProcessing
lda_model	LDA model, which is created from generate_lda using gensim.

`corpus`	Corpus for LDA, which is created using `doc_array` and `dict`
`prior`	Prior, which is updated later, after topic & word causality analysis and prior calculation.
`u`	Prior strength, which is defined in the main method and passed in during object initialization. Values can include: [10,50,100,500,1000]
`num_topics`	Number topics, which is defined in the main method and passed in during object initialization. Values can include: [10,20,30,40]

Class Procedure	Input	Output	Description
`set_corpus`	None	None	Generates `corpus`. Uses dict.doc2bow, `dict`, and each tokenized doc in `doc_array` to create the corpus for all NYT documents.
`generate_lda`	None	None	Generates `lda_model`. Uses gensim to generate corpus, using these parameters: `gensim.models.ldamodel.LdaModel(self.corpus, num_topics=self.num_topics, id2word=self.dict, passes=3, eta=self.prior, decay = self.u)`
`update_prior`	`new_prior`	None	Updates `prior` with `new_prior`, from later calculations, after topic & word causality analysis and prior calculation.
`get_top_words`	`topic_impacts`	`words`	Returns `words`, which is a list of top 10 words for each top topic in `topic_impacts`. `Topic_impacts` is a list of topic ids.

To generate the lda_model and get the top words, topics, and other probabilities we need for later analysis, we first `set_corpus` using the `dict` and `doc_array` to create the corpus for all documents. Then the LDA model is generated using gensim. Initially in the proposal, I tried refactoring the PLSA implementation from the MP assignments, but ended up using gensim after some research to ease integrating the background model. The other two methods are used later, where `update_prior` is used for time series feedback, and `get_top_words` is used to log the top words generated for each significant topic model.

2.4.3 Generate Top Topics with LDA

After we get the LDA model, we are able to determine the top topics. To do so, we get the topic information from `lda_model`, IEM time series data from `get_iem()`, dates from `dates`, and the significance value, which in this case is 0.95.

TopicCausality Class

Class Variable	Description
`lda_model`	LDA model, from LDA object
`iem_data`	Iem_data, from PreProcessing object
`dates`	Dates for each document, from LDA object
`sig`	Sig value, passed from main method. In this case, it's 0.95
`topic_ts`	List of time series data for each topic, used for Granger Causality. This is a 2-d list of length(# topics), with each index containing a list of summed topic coverages for each day. Format is: *[[topic1-day1 coverage sum, topic1-day2 coverage sum, etc...],* *[topic2-day1 coverage sum, topic2-day2 coverage sum], etc…],* *[etc...]]*
`topic_impacts`	List of impacts (p-val from Granger Causality) for each topic. This list is length(# topics), with each index referring to the topic's impact. This is used to pick the most significant topics for finding the top words for prior generation.

Class Procedure	Input	Output	Description
`generate_topic_ts`	None	None	Generates `topic_ts`. Uses `lda_model.get_document_topics(corpus)` to get topic coverage for each document. `Get_document_topics` returns a list of length(# documents), and for each document, you get (topic id, coverage). Using this info, we iterate through all documents and get their respective coverages, get the document's timestamp from `dates`, and update the current sum in the correct date index in the correct topic index in `topic_ts`.
`granger`	None	None	Generates `topic_impacts`. Uses iem and topic time series with 5 day lag to calculate the granger causality. In this case, we use the max p-value from the results. I tried averaging them across all 5 days, but using the max p-value yielded higher impact

			scores and improved the iterative topic model's performance.
`get_granger_avg`	None	Average Causality Confidence	Returns Average Causality Confidence by averaging all significance values from the Granger causality tests for all topics.

This class first is first initialized with the `lda_model, iem_data, sig. value, and dates.` Then, the time series data for each topic is calculated from `generate_topic_ts()` by iterating through all documents in `lda_model,` getting each of their topic coverages, and timestamp to sum up the document coverages for each topic, by timestamp, which in this case is each date. Next, we can use the granger causality method with a 5 day lag between the `iem_data` and `topic_ts` time series data. I tried averaging the p-values for each lag and compared the performance to picking the max p-value for each topic. I noticed that the max p-value was more likely to pass the 0.95 significance threshold, thus improving overall model performance, so I committed to selecting the max p-value. The averages would stay at around 0.7 and not help the model proceed. `get_granger_avg()` is used to log the Average Causality Confidence for the current iteration of topic model generation.

2.4.4 Generate Top Words from Top Topics

After running the procedures in TopicCausality, we now have the impacts for each topic. With this information, we can generate the top words from the top topics.

WordCausality Class

Class Variable	Description
`top_topics`	The entire TopicCausality from the previous step
`sig_topics`	List of significant topics (just the ids), used for prior generation
`documents`	List of paths to preprocessed NYT text files for each day (each line is pre-processed document), from `get_nyt` in PreProcessing class.
`dict`	Dictionary that maps each unique word/token in all docs to a unique int id, from `dict` in PreProcessing class.
`top_words`	List of length(# significant topics), where each index contains a list of the top 25 words.
`all_words`	List of all top 25 words for all topics
`word_count_stream`	Time series data for each word, which is the word count for all documents for each date in the time series.

word_correlations	List of Pearson Correlations between the word count stream and IEM time series, for each word in all_words

Class Procedure	Input	Output	Description
get_words	None	None	Generates sig_topics, top_words, and all_words. To do so, the function iterates through each topic in self.top_topics.topic_impacts and adds topics with an impact of 0.95 or greater to sig_topics. Then, it iterates through sig_topics to get the top 25 words for each topic, and adds them to top_words and all_words.
get_word_stream	None	None	Generates word_count_stream. The function iterates through all_words and counts the occurrences of the word in all docs for each date in the time series. This word count time series data is appended to word_count_stream for each word.
get_word_sig	None	None	Generates word_correlations, using the Pearson Correlation from the stats package to get the correlation between the word time series and IEM time series data. We append each word + correlation pair as 1 item in the word_correlations list.
get_word_correlations	None	Average Purity	Returns word correlations list, which is used for prior calculations.

After the top topics have been generated and the WordCausality object has been initialized with the appropriate variables, we first get the top 25 words that belong to each significant topic. The 25 value was chosen rather arbitrarily, but later during prior generation, the number of words are capped by the probM rule. This is done with the get_words() method. Once we know which words to conduct word count streams for, we call get_word_stream() to get the time series data for each word. Then, we call get_word_sig() to get the Pearson correlations between each word count stream and IEM time series data. Now, we have the significant values for each word from all the significant topics, so we are ready to generate the new prior.

2.4.5 Generate Prior from Top Words and Update Prior in LDA

Now that we have our significant topics and significant words with their impact values, we are ready to generate the prior.

PriorGeneration Class

Class Variable	Description
`top_words`	The entire WordCausality object from the previous step
`lda_model`	The entire LDA object from the previous step
`sig`	Significance value for prior calculation, which is set from the main method. Sig of 0.95 was used.
`probM`	Predefined constant set from the main method to cap the number of "top" words in the correlation list.
`pos_sorted_words`	List of positively correlated words for each topic, where the sum of all correlations (per topic) are <= `probM`
`neg_sorted_words`	List of negatively correlated words for each topic, where the sum of all correlations (per topic) are <= `probM`
`prior`	The prior, which will be used in the next iteration of the LDA model
`correlation_map`	A list where the word id is used as an index to get the word correlation/significance. This was used to speed up calculating the prior.
`purity`	Average purity, defined by section 5.1.1 in the paper.

Class Procedure	Input	Output	Description
`sort_words_per_topic`	None	None	Generates `pos_sorted_words`, `neg_sorted_words`, `correlation_map`, and `purity`. This function iterates through all the significant topics and figures out which significant words to add to the positive and negative impact words list. While doing so, it calculates the purity.
`calc_prior`	None	None	Generates the `prior`. Based on the formula in section 4.2.2 in the report, this prior first

			sums up all the significance values of each word for each topic (positive and negative are grouped separately) to get the divisor. Then, the function individually calculates the prior of each word. In the event that topics share the same top word, I chose the max prior.
`get_prior`	None	Prior	Returns `prior`

To calculate the prior, `sort_words_per_topic()` is called to figure out the positive and negative word groupings for each topic. It accomplishes this by first iterating through `word_correlations` and sorting which words have positive and negative correlations, putting them in `pos_words` and `neg_words` respectively. Then, the function figures out the positive and negative words that are in each topic while making sure the sum of the respective probabilities are less than `probM` by iterating through all words in each topic in order from highest to lowest probability, and checking if the word is in our either in `pos_words` or `neg_words`. If so, the words are appropriately put in `pos_sorted_words` and `neg_sorted_words`, where both lists are length(# significant topics), and each index contains the respective positively/negatively correlated words that make it past the `probM` cutoff. While doing so, it calculates the purity, since it already knows the number of positive and negatively correlated words.

Now that the positive and negative words have been determined for each topic, `calc_prior()` is called to calculate the prior for each word. In this function, first, the sums of all the correlations for each pos/neg grouping for each topic are calculated to get the divisor for the prior formula. Then, the function iterates through `pos_sorted_words` and `neg_sorted_words` to calculate the individual prior values. In the event that different topics contain the same word, the highest prior value is kept. Once the prior is calculated, it is stored as self.prior, which can later be retrieved to update the "eta" value in `gensim.models.ldamodel.LdaModel(...)`. The computer prior can be retrieved with `get_prior()`.

3. Results

3.1. Top Words in Significant Topics

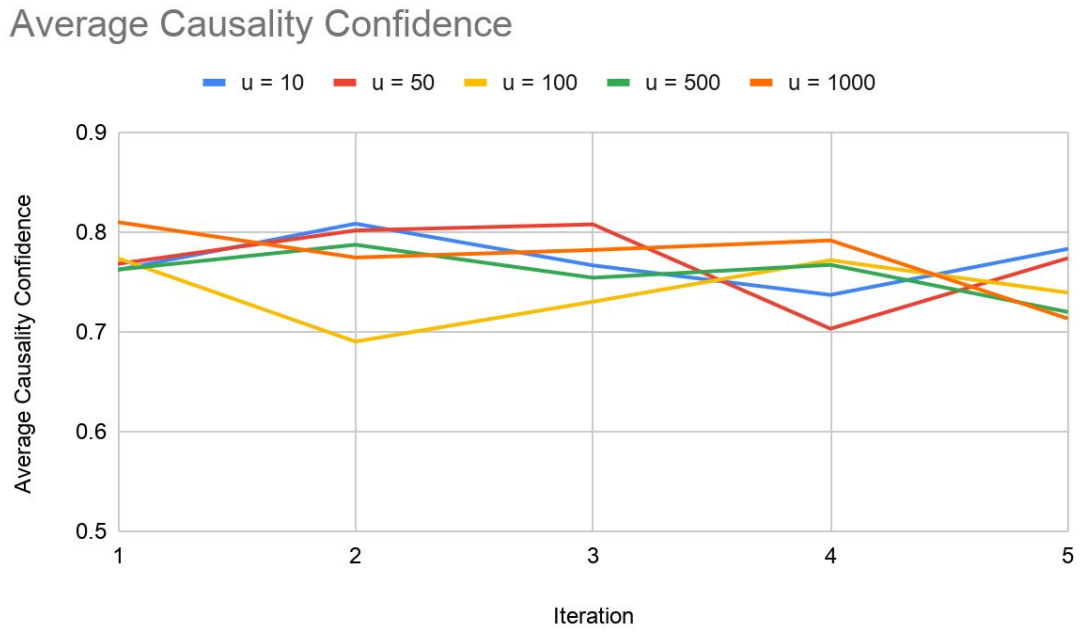
The data below used prior strength of u = 50, and a topic number of tn = 10, at its 2nd iteration, which is printed below:

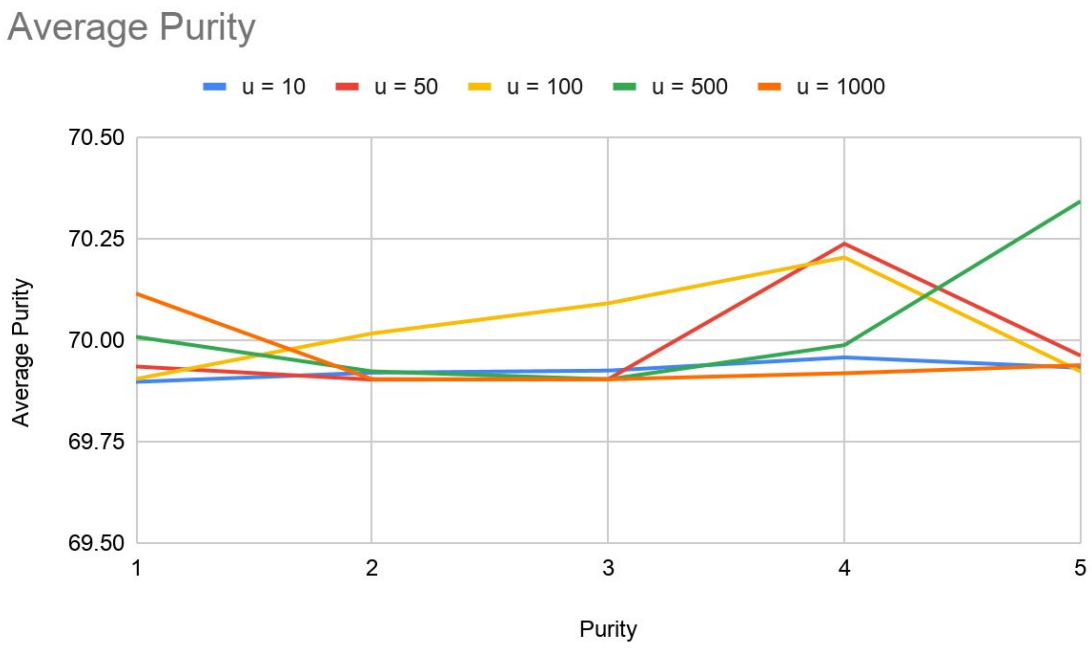
```
tn = 10
U = 50
Average Causality Confidence: 0.7449562561985322
Average Purity: 70.01112439977564
Top words:
```

```
s, w, gov, will, texas, people, security, new, time, party,
s, will, one, w, two, voters, tax, percent, new, democratic,
s, w, gov, new, will, one, also, voters, today, percent,
s, w, new, tax, gov, texas, people, can, plan, t,
s, w, one, party, gov, t, people, new, debate, today,
s, w, gov, new, will, one, plan, state, social, debate,
s, w, gov, will, abortion, republican, new, debate, today, texas,
s, w, republican, party, national, gov, one, t, will, texas,
s, w, gov, one, t, will, new, republican, m, convention,
s, w, gov, republican, texas, one, years, democratic, t, cheney,
s, will, w, texas, republican, gov, state, one, lieberman, democratic,
```

This result is notably different from that in the paper, though some of the words do make it and are colored red. The biggest reason for the difference may be due to the preprocessing. For scraping the data from the XML files, any data that was in the <p> tag was included, which may include text outside of the typical body of the article or be too strict and not include headlines and titles. Also, before the documents were tokenized, stop words were removed. On top of the stop words, specific words and phrases were also removed and replaced with an empty string, because they were dominating the results and did not provide useful information. These include the presidential candidate's names and their titles, the words "governor" and "gov", "clinton", "campaign", and "said". This might have messed up the tokenization and explain the prevalence of single letters "s, w, and t" in the results. Also, a **probM**, for the prior top words cutoff, was arbitrarily chosen to be 0.2. The selection of this number may have significantly thrown off the results. I was shocked at how low the probability of each word was, and the **probM** may have had to be lower.

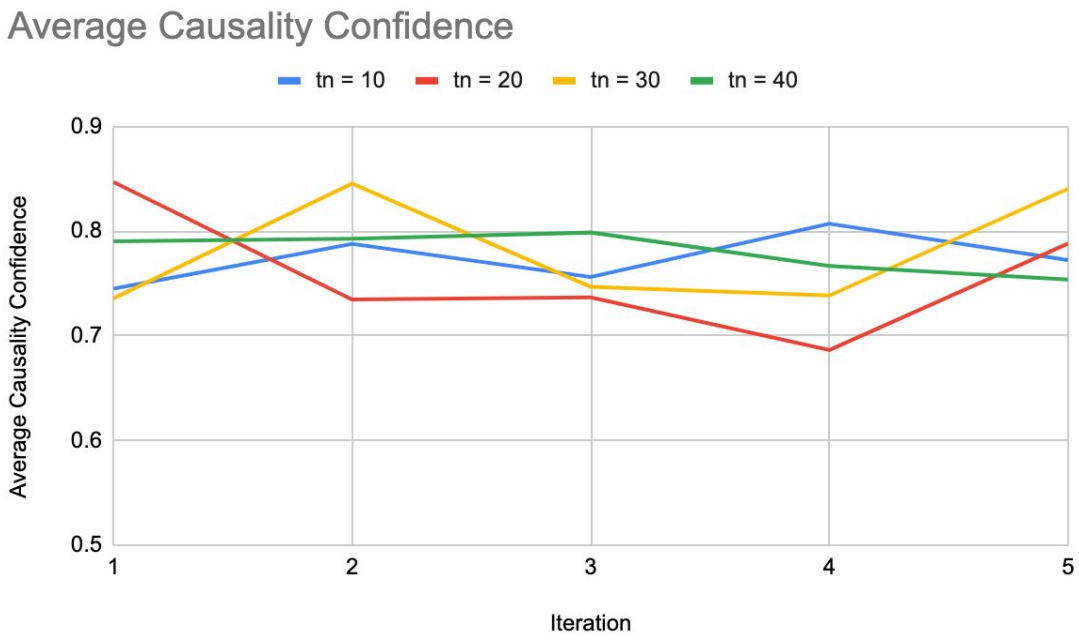
3.2 Different Prior Strengths and Topic Numbers

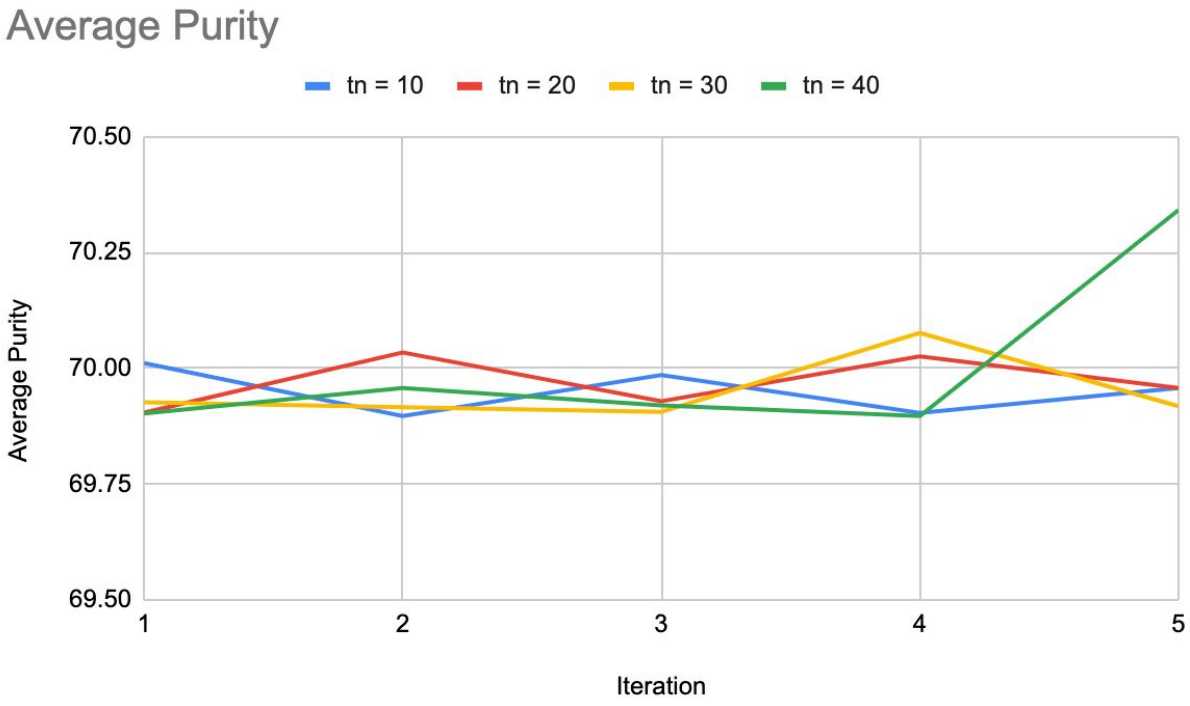




For changing the prior strength, some similarities include how the average causality confidence doesn't really correlate to the prior strength. However, the causality confidence does not increase

with the number of iterations, and for some prior strengths actually decrease. For the Purity, we do see that the runs with higher prior strength tend to have better purity, but the plots do end a bit differently towards the end, instead of leveling out in purity values from iteration 3 on, like in the paper.





Reasons for discrepancies:

Similar to the previous plots, changing the number of topics has not led to better performance and does not line up with those in the paper. Some reasons for the discrepancies may be due to the fact that I used a different LDA model (gensim) then that used in the paper. The preprocessing and word removal may have also been different for the NYT text data. In picking the topic impacts from Granger Causality results, I chose the max p-value out of all lags, which also may have been wrong. For calculating the purity, the logarithm I used was base 10, which may have been wrong. For the prior generation step, I randomly chose a **probM** value that may work, and the 0.2 value chosen may be incorrect. Also in the prior generation step, when multiple topics shared the same word, I took the highest prior value out of all the prior calculations, which also may have been wrong. For the solutions, I can experiment with different probM values to see if that parameter negatively impacted my prior generation.