Nestor

Toolkit Documentation

Thurston Sexton

None

Table of contents

1. Nestor

Machine-augmented annotation for technical text

downloads 18k code style black

You can do it; your machine can help.

1.1 Purpose

Nestor is a toolkit for using Natural Language Processing (NLP) with efficient user-interaction to perform structured data extraction with minimal annotation time-cost.

The Problem

NLP in technical domains requires context sensitivity. Whether for medical notes, engineering work-orders, or social/behavioral coding, experts often use specialized vocabulary with over-loaded meanings and jargon. This is incredibly difficult for off-the-shelf NLP systems to parse through.

The common solution is to contextualize and adapt NLP models to technical text -- Technical Language Processing (TLP) [1]. For instance, medical research has been greatly advanced with the advent of labeled, bio-specific datasets, which have domain-relevant named-entity tags and vocabulary sets. Unfortunately for analysts of these types of data, creating resources like this is incredibly time consuming. This is where `nestor` comes in.

Why Maintenance and Manufacturing?

A reader may notice a heavy focus on maintenance and manufacturing in the Nestor documentation and design. While this is a common problem in technical domains, generally, Nestor got its start in manufacturing data analysis. A large amount of maintenance data is *already* available for use in advanced manufacturing systems, but in a currently-unusable form: service tickets and maintenance work orders (MWOs).

For further reading, see [2] [3] [1].

1.2 Quick Links

- Get started
- Use a GUI
- Go to our Project Page

Nestor and all of it's associated gui's/projects are in the public domain (see the License). For more information and to provide feedback, please open an issue, submit a pull-request, or email us at nestor@nist.gov.

1.3 How does it work?

See the Getting Started page.

This application was originally designed to help manufacturers "tag" their maintenance work-order data according to the methods being researched by the Knowledge Extraction and Applications project at NIST. The goal is to help build context-rich labels in data sets that previously were too unstructured or filled with jargon to analyze. The current build is in very early alpha, so please be patient in using this application. If you have any questions, please do not hesitate to contact us (see Who are we?.)

- Rank keywords found in your data by importance, saving you time
- Suggest term unification by similarity (e.g. spelling), for quick review
- Basic entity relationship builder, to assist assembling problem code and taxonomy definitions
- Strucutred data output as named-entity tags, whether in readable (comma-sep) or computation-friendly (sparse-mat) form.

Planned:

- Customizable entity types and rules,
- Export to NER training formats,
- Command-line app and REST API.

1.4 Who are we?

This toolkit is a part of the Knowledge Extraction and Application for Smart Manufacturing (KEA) project, within the Systems Integration Division at NIST.

Projects that use Nestor

- Various Nestor GUIs: ways to use the full human-centered Nestor workflow in a user-interface.
- `nestor-eda` : (exploratory data analysis): things to do with Nestor-annotated data (dashboard, viz, etc.)

Points of Contact

- Email the development team at nestor@nist.gov
- Thurston Sexton @tbsexton Nestor Technical Lead, Associate Project Leader
- Michael Brundage Project Leader

Why "KEA"?

The KEA project seeks to better frame data collection and transformation systems within smart manufacturing as *collaborations* between human experts and the machines they partner with, to more efficiently utilize the digital and human resources available to manufacturers. Kea (*nestor notabilis*) on the other hand, are the world's only alpine parrots, finding their home on the southern Island of NZ. Known for their intelligence and ability to solve puzzles through the use of tools, they will often work together to reach their goals, which is especially important in their harsh, mountainous habitat.

1.5 Development/Contribution Guidelines

More to come, but primary requirement is the use of Poetry. Plugins are installed as development dependencies through poetry (e.g. `taskipy` and `poetry-dynamic-versioning`), though if not using `conda` environments, `poetry-dynamic-versioning` may require being installed to the global python installation.

Notebooks should be kept nicely git-friendly with Jupytext

1.6 Other Tools/Resources

Know of other tools? Or want to find similar resources as Nestor? A community driven TLP Community of Interest (COI) has been created to provide publicly available resources to the community. Check out our awesomelist.

---

1. Michael P Brundage, Thurston Sexton, Melinda Hodkiewicz, Alden Dima, and Sarah Lukens. Technical language processing: unlocking maintenance knowledge. *Manufacturing Letters*, 2020. ↩↩
2. Thurston Sexton, Michael P Brundage, Michael Hoffman, and Katherine C Morris. Hybrid datafication of maintenance logs from ai-assisted human tags. In *Big Data Big Data, 2017 IEEE International Conference on*, 1769–1777. IEEE, 2017. ↩
3. Michael Sharp, Thurston Sexton, and Michael P Brundage. Toward semi-autonomous information. In *IFIP International Conference on Advances in Production Management Systems*, 425–432. Springer, 2017. ↩

2. License/Terms-of-Use

2.1 Software Disclaimer / Release

This software was developed by employees of the National Institute of Standards and Technology (NIST), an agency of the Federal Government and is provided to you as a public service. Pursuant to title 15 United States Code Section 105, works of NIST employees are not subject to copyright protection within the United States.

The software is provided by NIST "AS IS." NIST MAKES NO WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT AND DATA ACCURACY. NIST does not warrant or make any representations regarding the use of the software or the results thereof, including but not limited to the correctness, accuracy, reliability or usefulness of the software.

To the extent that NIST rights in countries other than the United States, you are hereby granted the non-exclusive irrevocable and unconditional right to print, publish, prepare derivative works and distribute the NIST software, in any medium, or authorize others to do so on your behalf, on a royalty-free basis throughout the World.

You may improve, modify, and create derivative works of the software or any portion of the software, and you may copy and distribute such modifications or works. Modified works should carry a notice stating that you changed the software and should note the date and nature of any such change.

You are solely responsible for determining the appropriateness of using and distributing the software and you assume all risks associated with its use, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and the unavailability or interruption of operation. This software is not intended to be used in any situation where a failure could cause risk of injury or damage to property.

Please provide appropriate acknowledgments of NIST's creation of the software in any copies or derivative works of this software.

2.2 3rd-Party Endorsement Disclaimer

The use of any products described in this toolkit does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that products are necessarily the best available for the purpose.

3. Getting Started

3.1 Getting Started

To install `nestor`, utilize a python installation (preferrably an environment like `pyenv` or `conda`) to install from the Pypi repository:

```
pip install nist-nestor
```

The core `nestor` module is intended to assist analysts (and the UIs or pipelines the may create) in annotating technical text. If you just want to jump in to a more polished experience, head over to the User Interfaces page.

3.2 Getting Started

Motivation

NLP in technical domains often involves

1. Narrowing down the set of concepts written about to a subset of *relevant*, well-defined, possibly related ones;
2. Annotating whether/where those concepts occur within documents of a dataset, for training or validation purposes.

This can take the form of named entity recognition (NER) training sets, lists of domain-specific stopwords, etc. In medical NLP, this often uses agreed-upon named-entity sets that have been added *by-hand* to several corpuses. Check out the scispacy page from Allen AI for an example: one set uses `disease` and `chemical`, another uses `dna`, `protein`, `cell-type`, `cell-line`, and `rna`, etc.

Creating the set of concepts in a way that enough people can agree on a method for tagging their occurrence is incredibly time-consuming. Often, an analyst needs to iterate and prototype various entity sets, and test their coverage and usefulness in a specific task. This analyst needs a way to rapidly estimate the sets of entities --- the "tags" they will use --- that meaningfully represent the data at hand. The insight we bring to bear is that this does not need to be done *document-by-document* in order to be data-driven.

3.3 Getting Started

Workflow

```
graph LR
  text --> keywords
  keywords --> priorities
  keywords --> relationships
  priorities --> user
  relationships --> user
  user -->|types+normalization| keywords
```

Core Idea

Nestor starts out with a dataset of records with certain columns containing text. This text is cleaned up and scanned for **keywords** that are statistically important to the corpus (e.g. using sum-tfidf; see [1]).

This gives an analyst an overview of terms that happen often or in special contexts, and must be dealt with as relevant (or not) to their analyses. Especially important are the term **priorities**: as one travels further down the list, terms are deemed less important *to the machine*, giving the user a sense of how algorithms are "seeing" the corpus.

However, in techical text, shorthands and jargon quickly develop. What started out being called "hydraulic" or "air conditioning unit" may eventually be called "hyd" or "ACU", obscuring statistical significance of both. These **relationships** can be determined through similarity in a number of ways: in our experience a useful similarity is to use variations on *Levenstein distance* to catch misspellings or abbreviations, but there are many others.

Both the keyterms and their relationships can now be passed to a user for them to **type** as needed, structuring the now-named-entities as needed. Importantly, this is done in order of percieved importance, minimizing wasted time!

> *types* available are determined by `entities` property of the `nestor.CFG` configuration object, which is set to use a maintenance-centric entity type system by default (`problem`, `item`, `solution`). Future releases will allow customization of this list!

The "too-hot" problem

Often a keyword won't make sense out-of-context: if "hot" is important to an HVAC maintenance dataset, it is definitely important! But it may refer to a room being "too hot" (a problem), or perhaps "hot water" (just an object). This means the user can't know what to *type* the "hot" entity without more context.

Nestor uses the idea of *derived types*, so that context-sensitive keywords can be built out of otherwise ambiguous blocks.

> Derived types are governed by the `derived` property of `nestor.CFG`. Rules for creating them from atomic types are defined by the `entity_rules_map` property. See `nestor.settings` for more information.

---

1. Franziska Horn, Leila Arras, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. Exploring text datasets by visualizing relevant words. *arXiv preprint arXiv:1707.05261*, 2017. ↩

4. Nestor Workflow as a Graphical User Interface

Intended as a user-centric workflow, many of the associated tasks are better approached through a GUI. Nestor originally began as a prototype interface to a scikit-learn pipeline. It morphed a bit over time, as we realized that consistent iteration on existing annotations with feedback from the algorithms were crucial.

There are several interfaces under development:

- `nestor-qt` : legacy desktop-oriented app
- most development and feature-complete
- no longer actively developed (still maintained)
- `nestor-web`
- modern interface built on Electron
- fewer features; being actively developed

5. Examples

5.1 Case Study: Excavator Survival Analysis

Mining Excavator dataset case study, as originally presented in Sexton et al. [^1].

```
from pathlib import Path
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import nestor
from nestor import keyword as kex
import nestor.datasets as dat
def set_style():
    """This sets reasonable defaults for a figure that will go in a paper"""
    sns.set_context("paper")
    sns.set(font='serif')
    sns.set_style("white", {
        "font.family": "serif",
        "font.serif": ["Times", "Palatino", "serif"]
    })
set_style()
```

```
df = dat.load_excavators()
df.head()
```

	BscStartDate	Asset	OriginalShorttext	PMType	Cost
ID					
0	2004-07-01	A	BUCKET WON'T OPEN	PM01	183.05
1	2005-03-20	A	L/H BUCKET CYL LEAKING.	PM01	407.40
2	2006-05-05	A	SWAP BUCKET	PM01	0.00
3	2006-07-11	A	FIT BUCKET TOOTH	PM01	0.00
4	2006-11-10	A	REFIT BUCKET TOOTH	PM01	1157.27

```
vocab = dat.load_vocab('excavators')
vocab
```

	NE	alias	notes	score
tokens				
replace	S	replace	NaN	0.033502
bucket	I	bucket	NaN	0.018969
repair	S	repair	NaN	0.017499
grease	I	grease	NaN	0.017377
leak	P	leak	NaN	0.016591
...	...	...	...	...
1boily 19	NaN	NaN	NaN	0.000046
shd 1fitter	NaN	NaN	NaN	0.000046
19 01	NaN	NaN	NaN	0.000046
01 10	NaN	NaN	NaN	0.000046
1fitter 1boily	NaN	NaN	NaN	0.000046

6767 rows × 4 columns

Knowledge Extraction

We already have vocabulary and data, so let's merge them to structure our data to a more useful format.

scikit-learn's `Pipeline`

Convenient way to use `TagExtractor` to output a more usable format. Let's use the multi-index `binary` format for now. Other options include: - list-of-tokens `multilabel` - NER-trainer `iob`.

```
# merge and cleanse NLP-containing columns of the data
from sklearn.pipeline import Pipeline

pipe = Pipeline([
    ('clean_combine', kex.NLPSelect(columns=["OriginalShorttext"])),
    ('tag', kex.TagExtractor(
        thesaurus=vocab, # load up pre-trained vocab file from [^1]
        group_untagged=True,  # merge untagged tokens into a misc. `_untagged` label
        filter_types=None,  # keep all tag types
        ngram_range=(1,2),  # this vocab file includes compound tags, so make sure we find them
        verbose=True,  # report how much coverage our vocab file got on proposed tags (tokens)
        output_type='binary', # could use `multilabel` or `iob` as well
    )),
])
tags = pipe.fit_transform(df)

tags.sum().sort_values(ascending=False)
```

```
100%|████████████████████████████████████████████████████████████| 8/8 [00:02<00:00,  3.60it/s]
```

```
Complete Docs: 1402, or 25.56%
Tag completeness: 0.72 +/- 0.21
Empty Docs: 47, or 0.86%
```

```
NA  _untagged          8006.0
S   replace            1364.0
P   leak                748.0
I   engine              603.0
S   repair              584.0
                        ...
SI  replace battery       1.0
    replace clamp         1.0
PI  ring leak             1.0
I   step_handrail         1.0
    windscreen_wiper      1.0
Length: 1011, dtype: float64
```

We can also access the trained steps in the pipeline to make use of convenience functions.

```
tagger = pipe.named_steps['tag']
tags_read = tagger.tags_as_lists

relation_df = tags.loc[:, ['PI', 'SI']]
tag_df = tags.loc[:, ['I', 'P', 'S', 'U', 'X', 'NA']]
```

Quality of Extracted Keywords

It's

```
nbins = int(np.percentile(tag_df.sum(axis=1), 90))
print(f'Docs have at most {nbins} tokens (90th percentile)')
```

```
Docs have at most 9 tokens (90th percentile)
```

```
tags_read.assign(text=df.OriginalShorttext).sample(10)
```

	I	NA	P	PI	S	SI	U	X	text
4890	cable		emergency, need		pull, replace				emergency pull cable needs replacing
4947	camera	_untagged	damage		repair				REPAIR 3 DAMAGED CAMERAS.
3496	timer						idle, working		Idle timer not working
3168	grease, link, link_pin, pin, shd	_untagged							No grease to bottom H-Link Pin SHD0024
544	bucket		cracked		repair				REPAIR CRACKS IN BUCKET
614	bucket, bucket_grease, grease, line	_untagged	broken	broken bucket					2 X BROKEN BUCKET GREASE LINES
752	engine, left_hand		blowing, smoke						lh eng blowing smoke
2908	grease, rotor, steel, steel_tube, tube		leak						STEEL TUBE IN ROTOR LEAKING GREASE
695	air				cleaners, replace	air cleaners, replace air			replace air cleaners
3978	boom, boom_cylinder, cylinder				replace	replace boom			replace r/h boom cylinder

```
# how many instances of each keyword class are there?
print('named entities: ')
print('I\tItem\nP\tProblem\nS\tSolution')
print('U\tUnknown\nX\tStop Word')
print('total tokens: ', vocab.NE.notna().sum())
print('total tags: ', vocab.groupby("NE").nunique().alias.sum())
vocab.groupby("NE").nunique()
```

```
named entities:
I   Item
P   Problem
S   Solution
U   Unknown
X   Stop Word
total tokens:  4060
total tags:  1123
```

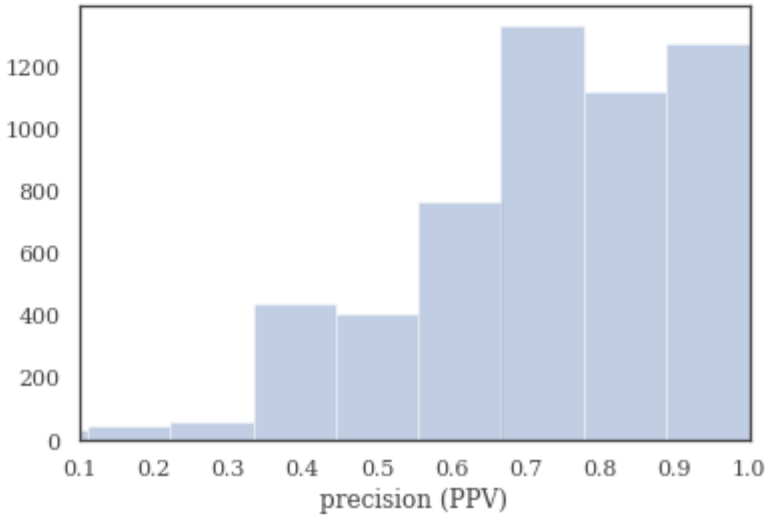
	alias	notes	score
NE			
I	633	18	1856
P	55	5	122
PI	150	0	672
S	44	1	97
SI	134	0	446
U	68	56	92
X	39	0	167

```
# tag-completeness of work-orders?
tagger.report_completeness()

# with sns.axes_style('ticks') as style:
sns.distplot(tagger.tag_completeness.dropna(),
             kde=False, bins=nbins,
             kde_kws={'cut':0})
plt.xlim(0.1, 1.0)
plt.xlabel('precision (PPV)')
```

```
Complete Docs: 1402, or 25.56%
Tag completeness: 0.72 +/- 0.21
Empty Docs: 47, or 0.86%
```

```
/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

```
Text(0.5, 0, 'precision (PPV)')
```



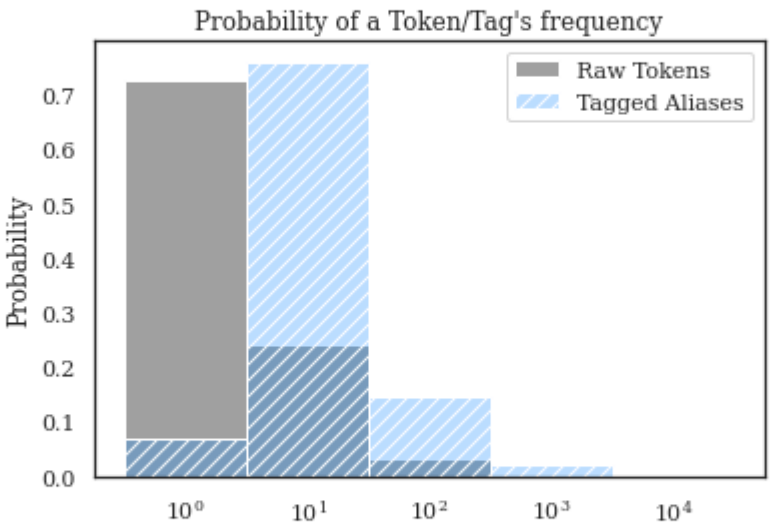
An Aside: Effectiveness of Tagging

What have we actually gained using the `TagExtractor`?

The `vocab` file functions as a thesaurus, that has a default `alias` representing multiple disparate tokens. This means our resulting matrix dimensionality can be significantly reduced *using this domain knowledge*, which can improve model predictability, performance, applicability, etc.

```python
# original token string frequencies
cts = np.where(tagger.tfidf.todense()>0., 1, 0).sum(axis=0)
sns.histplot(
    cts, log_scale=True,
    stat='probability',discrete=True,
    label='Raw Tokens',
    color='grey'
)
# tag frequencies
sns.histplot(
    tag_df[['I', 'P', 'S']].sum(), log_scale=True,
    stat='probability', discrete=True,
    label='Tagged Aliases',
    hatch='///',
    color='dodgerblue',alpha=0.3,
)
plt.legend()
plt.title('Probability of a Token/Tag\'s frequency')
```

```
Text(0.5, 1.0, "Probability of a Token/Tag's frequency")
```



The entire goal, in some sense, is for us to remove low-occurence, unimportant information from our data, and form concept conglomerates that allow more useful statistical inferences to be made. Tags mapped from `nestor-gui`, as the plot shows, have very few instances of 1x-occurrence concepts, compared to several thousand in the raw-tokens (this is by design, of course). Additionally, high occurence concepts that might have had misspellings or synonyms drastically inprove their average occurence rate.

NOTE: This is *without* artificial thresholding of minimum tag frequency. This would simply get reflected by "cutting off" the blue distribution below some threshold, not changing its shape overall.

Survival Analysis

What do we *do* with tags?

One way is to rely on their ability to normalize raw tokens into consistent aliases so that our estimates of rare-event statistics become possible.

Say you wish to know the median-time-to-failure of an excavator subsystem (e.g. the engines in your fleet): this might help understand the frequency "engine expertise" is needed to plan for hiring or shift scheduls, etc.

To get the raw text occurences into something more consistent for failure-time estimation, one might:

- make a rules-based algorithm that checks for known (a priori) pattern occurrences and categorizes/normalizes when matched (think: Regex matching)
- create aliases for raw tokens (e.g. using suggestions for "important" tokens from TokenExtractor.thesaurus_template)

This was done in [^1], and whe demonstrate the technique below. See the paper for further details!

Rules-Based

From Hodkeiwiz et al, a rule-based method was used to estimate failure times for SA. Let's see their data:

```
df_clean = dat.load_excavators(cleaned=True)

df_clean['SuspSugg'] = pd.to_numeric(df_clean['SuspSugg'], errors='coerce')
df_clean.dropna(subset=['RunningTime', 'SuspSugg'], inplace=True)

df_clean.shape
```

```
(5289, 16)
```

```
df_clean.sort_values('BscStartDate').head(10)
```

ID	BscStartDate	Asset	OriginalShorttext	PMType	Cost	RunningTime	MajorSystem	Part	Action
8	2001-07-19	B	REPLACE LIP	PM01	1251.52	7.0	Bucket	NaN	I
1820	2001-09-01	B	OIL LEAK L/H TRACK TENSIONER.	PM01	0.00	3.0	Hydraulic System	Track	Mino
1821	2001-09-04	B	BAD SOS METAL IN OIL	PM01	0.00	3.0	Hydraulic System	Slew Gearbox	
5253	2001-09-05	B	REPLACE AIRCONDITIONER BELTS	PM01	0.00	23.0	NaN	Air Conditioning	I
3701	2001-09-05	B	REPLACE CLAMPS ON CLAM PIPES	PM01	0.00	28.0	NaN	Mount	I
1167	2001-09-05	B	REPLACE RHS FAN BELT TENSIONER PULLEY	PM01	82.09	0.0	NaN	Fan	Maint_I
1168	2001-09-11	B	replace fan belt	PM01	0.00	6.0	NaN	Fan	I
644	2001-09-15	B	replace heads on lhs eng	PM01	0.00	33.0	Engine	NaN	I
4583	2001-09-26	B	REPAIR CABIN DOOR FALLING OFF.	PM01	0.00	27.0	NaN	Drivers Cabin	
9	2001-10-01	B	rebuild lip #3	PM01	0.00	74.0	Bucket	NaN	

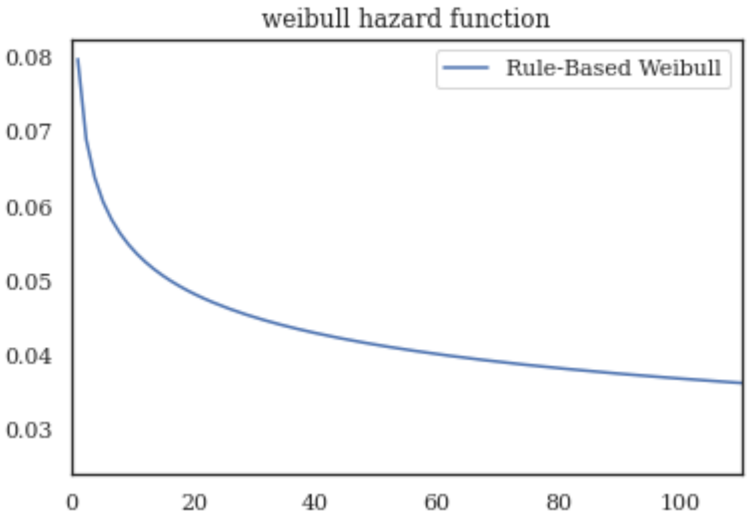
We once again turn to the library Lifelines as the work-horse for finding the Survival function (in this context, the probability at time t since the previous MWO that a new MWO has **not** occured).

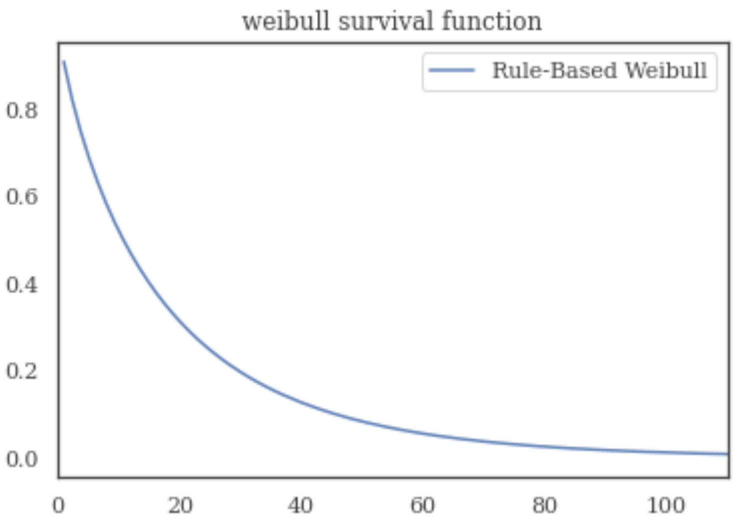
```
from lifelines import WeibullFitter, ExponentialFitter, KaplanMeierFitter
mask = (df_clean.MajorSystem =='Bucket')
# mask=df_clean.index
def mask_to_ETclean(df_clean, mask, fill_null=1.):
    filter_df = df_clean.loc[mask]
    g = filter_df.sort_values('BscStartDate').groupby('Asset')
    T = g['BscStartDate'].transform(pd.Series.diff).dt.days
#     T.loc[(T<=0.)|(T.isna())] = fill_null
    E = (~filter_df['SuspSugg'].astype(bool)).astype(int)
    return T.loc[~((T<=0.)|(T.isna()))], E.loc[~((T<=0.)|(T.isna()))]

T, E = mask_to_ETclean(df_clean, mask)
wf = WeibullFitter()
wf.fit(T, E, label='Rule-Based Weibull')
```

```python
print('{:.3f}'.format(wf.lambda_), '{:.3f}'.format(wf.rho_))
# wf.print_summary()
wf.hazard_.plot()
plt.title('weibull hazard function')
plt.xlim(0,110)

wf.survival_function_.plot()
plt.xlim(0,110)
plt.title('weibull survival function')
print(f'transform: β={wf.rho_:.2f}\tη={1/wf.lambda_:.2f}')
# wf._compute_standard_errors()
to_bounds = lambda row:'±'.join([f'{i:.2g}' for i in row])
wf.summary.iloc[:,:2].apply(to_bounds, 1)
```

```
16.733 0.833
transform: β=0.83   η=0.06
```

```
lambda_       17±0.94
rho_       0.83±0.026
dtype: object
```





Tag Based Comparison

We estimate the occurence of failures with tag occurrences.

```python
import math
```

```
def to_precision(x,p):
    """
    returns a string representation of x formatted with a precision of p

    Based on the webkit javascript implementation taken from here:
    https://code.google.com/p/webkit-mirror/source/browse/JavaScriptCore/kjs/number_object.cpp
    """

    x = float(x)

    if x == 0.:
        return "0." + "0"*(p-1)

    out = []

    if x < 0:
        out.append("-")
        x = -x

    e = int(math.log10(x))
    tens = math.pow(10, e - p + 1)
    n = math.floor(x/tens)

    if n < math.pow(10, p - 1):
        e = e -1
        tens = math.pow(10, e - p+1)
        n = math.floor(x / tens)

    if abs((n + 1.) * tens - x) <= abs(n * tens -x):
        n = n + 1

    if n >= math.pow(10,p):
        n = n / 10.
        e = e + 1

    m = "%.*g" % (p, n)

    if e < -2 or e >= p:
        out.append(m[0])
        if p > 1:
            out.append(".")
            out.extend(m[1:p])
        out.append('e')
        if e > 0:
            out.append("+")
        out.append(str(e))
    elif e == (p -1):
        out.append(m)
    elif e >= 0:
        out.append(m[:e+1])
        if e+1 < len(m):
            out.append(".")
            out.extend(m[e+1:])
    else:
        out.append("0.")
        out.extend(["0"]*-(e+1))
        out.append(m)

    return "".join(out)

def query_experiment(name, df, df_clean, rule, tag, multi_tag, prnt=False):

    def mask_to_ETclean(df_clean, mask, fill_null=1.):
        filter_df = df_clean.loc[mask]
        g = filter_df.sort_values('BscStartDate').groupby('Asset')
        T = g['BscStartDate'].transform(pd.Series.diff).dt.days
        E = (~filter_df['SuspSugg'].astype(bool)).astype(int)
        return T.loc[~((T<=0.)|(T.isna()))], E.loc[~((T<=0.)|(T.isna()))]

    def mask_to_ETraw(df_clean, mask, fill_null=1.):
        filter_df = df_clean.loc[mask]
        g = filter_df.sort_values('BscStartDate').groupby('Asset')
        T = g['BscStartDate'].transform(pd.Series.diff).dt.days
        T_defined = (T>0.)|T.notna()
        T = T[T_defined]
        # assume censored when parts replaced (changeout)
        E = (~(tag_df.S.changeout>0)).astype(int)[mask]
        E = E[T_defined]
        return T.loc[~((T<=0.)|(T.isna()))], E.loc[~((T<=0.)|(T.isna()))]

    experiment = {
        'rules-based': {
            'query': rule,
            'func': mask_to_ETclean,
            'mask': (df_clean.MajorSystem == rule),
            'data': df_clean
        },
        'single-tag': {
            'query': tag,
            'func': mask_to_ETraw,
            'mask': tag_df.I[tag].sum(axis=1)>0,
            'data': df
        },
```

```
        'multi-tag': {
            'query': multi_tag,
            'func': mask_to_ETraw,
            'mask': tag_df.I[multi_tag].sum(axis=1)>0,
            'data': df
        }
    }
    results = {
       ('query', 'text/tag'): [],
#        ('Weibull Params', r'$\lambda$'): [],
       ('Weibull Params', r'$\beta$'): [],
       ('Weibull Params', '$\eta$'): [],
       ('MTTF', 'Weib.'): [],
       ('MTTF', 'K-M'): []
       }
    idx = []

    for key, info in experiment.items():
        idx += [key]
        results[('query','text/tag')] += [info['query']]
        if prnt:
            print('{}: {}'.format(key, info['query']))
        info['T'], info['E'] = info['func'](info['data'], info['mask'])
        wf = WeibullFitter()
        wf.fit(info['T'], info['E'], label=f'{key} weibull')

        to_bounds = lambda row:'$\pm$'.join([to_precision(row[0],2),
                                             to_precision(row[1],1)])

        params = wf.summary.T.iloc[:2]
        params['eta_'] = [1/params.lambda_['coef'],  # err. propagation
                          (params.lambda_['se(coef)']/params.lambda_['coef']**2)]
        params = params.T.apply(to_bounds, 1)

        results[('Weibull Params', r'$\eta$')] += [params['eta_']]
        results[('Weibull Params', r'$\beta$')] += [params['rho_']]
        if prnt:
            print('\tWeibull Params:\n',
                  '\t\tη = {}\t'.format(params['eta_']),
                  'β = {}'.format(params['rho_']))

        kmf = KaplanMeierFitter()
        kmf.fit(info["T"], event_observed=info['E'], label=f'{key} kaplan-meier')
        results[('MTTF','Weib.')] += [to_precision(wf.median_survival_time_,3)]
        results[('MTTF','K-M')] += [to_precision(kmf.median_survival_time_,3)]
        if prnt:
            print(f'\tMTTF: \n\t\tWeib \t'+to_precision(wf.median_survival_time_,3)+\
                   '\n\t\tKM \t'+to_precision(kmf.median_survival_time_,3))
        info['kmf'] = kmf
        info['wf'] = wf
    return experiment, pd.DataFrame(results, index=pd.Index(idx, name=name))
```

```
bucket_exp, bucket_res = query_experiment('Bucket', df, df_clean,
                                          'Bucket',
                                          ['bucket'],
                                          ['bucket', 'tooth', 'lip', 'pin']);
```

```
tags = ['hyd', 'hose', 'pump', 'compressor']
hyd_exp, hyd_res = query_experiment('Hydraulic System', df, df_clean,
                                    'Hydraulic System',
                                    ['hyd'],
                                    tags)
```

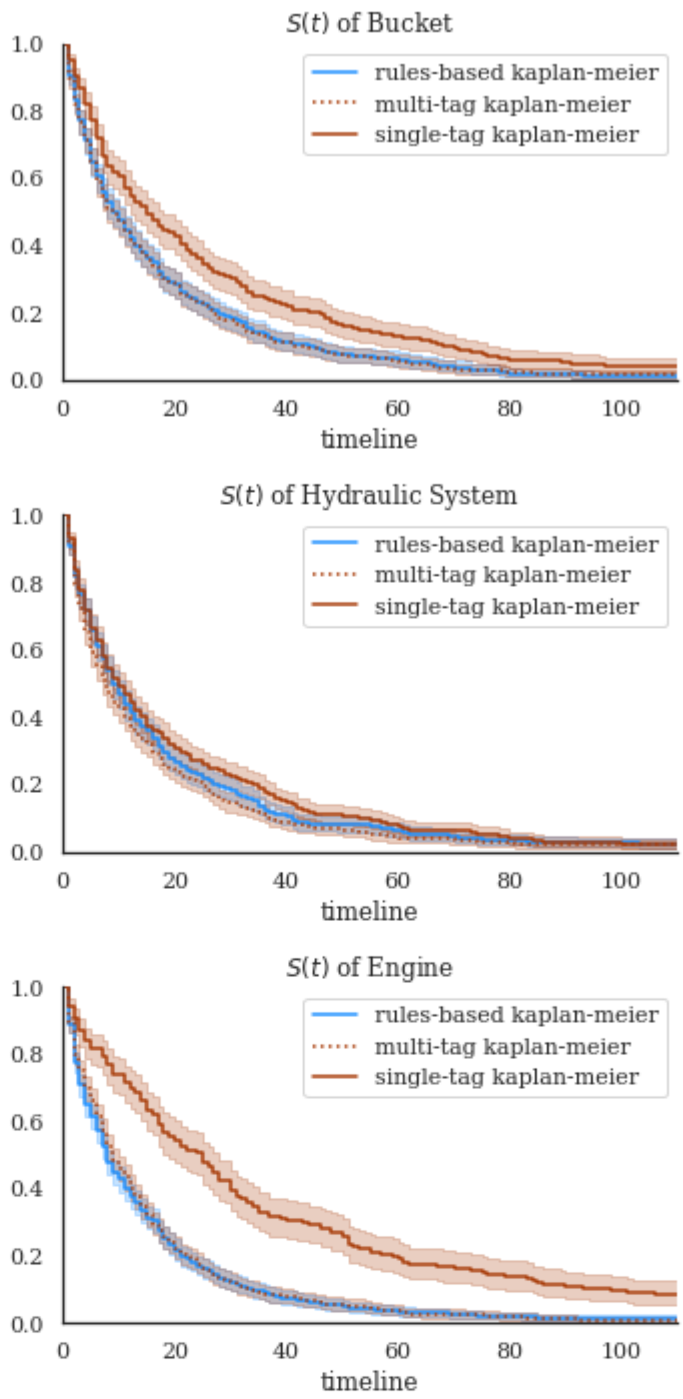
```
eng_exp, eng_res = query_experiment('Engine', df, df_clean,
                                    'Engine',
                                    ['engine'],
                                    ['engine', 'filter', 'fan'])
```

```
frames = [bucket_res, hyd_res, eng_res]
res = pd.concat(frames, keys = [i.index.name for i in frames],
```

```
               names=['Major System', 'method'])
res
```

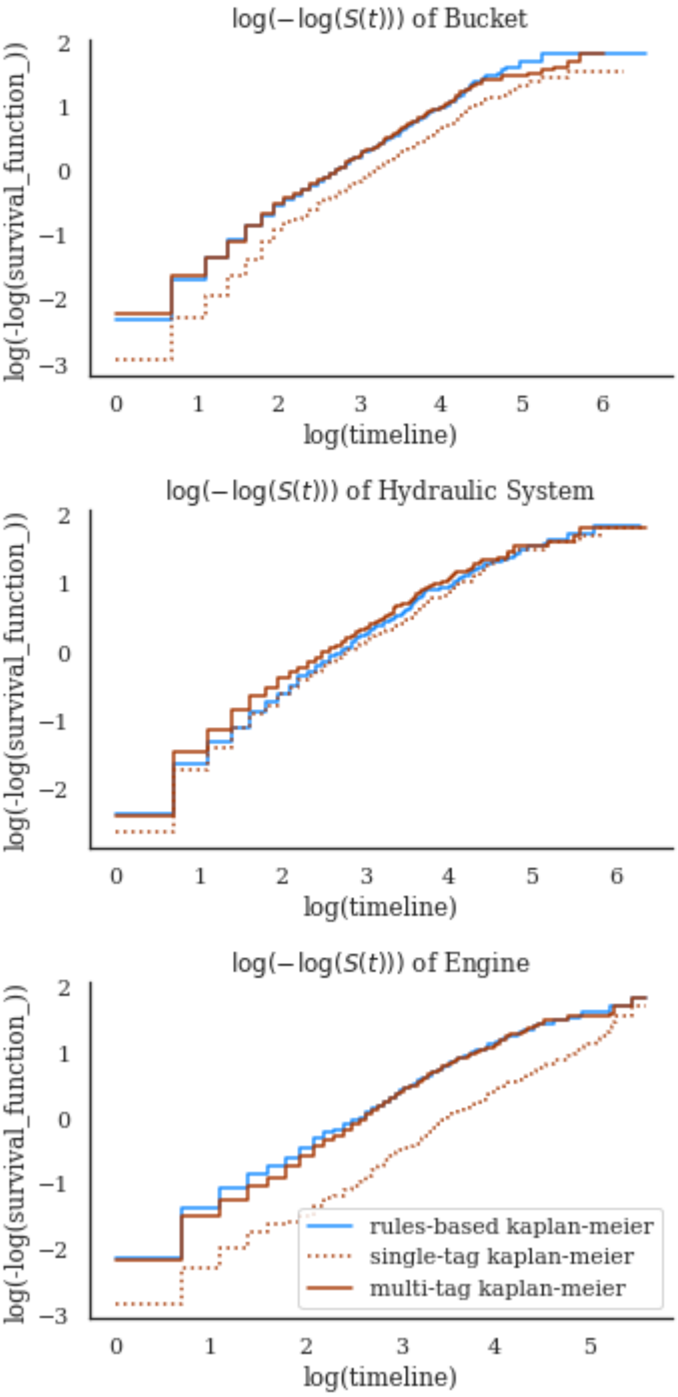
		query	Weibull Params		MTTF	
		text/tag	β	η	Weib.	K-M
Major System	**method**					
Bucket	rules-based	Bucket	0.83$\pm$0.03	0.060$\pm$3e-3	10.8	9.00
	single-tag	[bucket]	0.83$\pm$0.03	0.038$\pm$3e-3	17.0	15.0
	multi-tag	[bucket, tooth, lip, pin]	0.82$\pm$0.02	0.060$\pm$3e-3	10.6	9.00
Hydraulic System	rules-based	Hydraulic System	0.86$\pm$0.02	0.072$\pm$3e-3	9.02	8.00
	single-tag	[hyd]	0.89$\pm$0.04	0.027$\pm$2e-3	24.3	25.0
	multi-tag	[hyd, hose, pump, compressor]	0.88$\pm$0.02	0.068$\pm$3e-3	9.71	9.00
Engine	rules-based	Engine	0.81$\pm$0.02	0.059$\pm$3e-3	10.8	9.00
	single-tag	[engine]	0.80$\pm$0.03	0.053$\pm$3e-3	12.0	10.0
	multi-tag	[engine, filter, fan]	0.81$\pm$0.02	0.068$\pm$4e-3	9.31	8.00

```
exp = [bucket_exp, eng_exp, hyd_exp]
f,axes = plt.subplots(nrows=3, figsize=(5,10))
for n, ax in enumerate(axes):
    exp[n]['rules-based']['kmf'].plot(ax=ax, color='dodgerblue')
    exp[n]['multi-tag']['kmf'].plot(ax=ax, color='xkcd:rust', ls=':')
    exp[n]['single-tag']['kmf'].plot(ax=ax, color='xkcd:rust')

    ax.set_xlim(0,110)
    ax.set_ylim(0,1)
    ax.set_title(r"$S(t)$"+f" of {res.index.levels[0][n]}")
    sns.despine()
plt.tight_layout()
```



This next one give you an idea of the differences better. using a log-transform. the tags under-estimate death rates a little in the 80-130 day range, probably because there's a failure mode not captured by the [bucket, lip, tooth] tags (because it's rare).

```
f,axes = plt.subplots(nrows=3, figsize=(5,10))
for n, ax in enumerate(axes):
    exp[n]['rules-based']['kmf'].plot_loglogs(ax=ax, c='dodgerblue')
    exp[n]['single-tag']['kmf'].plot_loglogs(ax=ax, c='xkcd:rust', ls=':')
    exp[n]['multi-tag']['kmf'].plot_loglogs(ax=ax, c='xkcd:rust')
    if n != 2:
        ax.legend_.remove()
#     ax.set_xlim(0,110)
#     ax.set_ylim(0,1)
    ax.set_title(r"$\log(-\log(S(t)))$"+f" of {res.index.levels[0][n]}")
    sns.despine()
plt.tight_layout()
f.savefig('bkt_logKMsurvival.png')
# kmf.plot_loglogs()
```



$\log(-\log(S(t)))$ of Bucket
log(-log(survival_function_))
log(timeline)
$\log(-\log(S(t)))$ of Hydraulic System
log(-log(survival_function_))
log(timeline)
$\log(-\log(S(t)))$ of Engine
log(-log(survival_function_))
log(timeline)
rules-based kaplan-meier
single-tag kaplan-meier
multi-tag kaplan-meier

5.2 Named Entity Recognition

Output tags in IOB format for NER analysis

```
import pandas as pd
from pathlib import Path
from nestor import keyword as kex
import nestor.datasets as nd
```

```
# Get raw MWOs
df = (nd.load_excavators(cleaned=False)  # already formats dates
#       .rename(columns={'BscStartDate': 'StartDate'})
      )

# Change date column to DateTime objects
df.head(5)
```

ID	BscStartDate	Asset	OriginalShorttext	PMType	Cost
0	2004-07-01	A	BUCKET WON'T OPEN	PM01	183.05
1	2005-03-20	A	L/H BUCKET CYL LEAKING.	PM01	407.40
2	2006-05-05	A	SWAP BUCKET	PM01	0.00
3	2006-07-11	A	FIT BUCKET TOOTH	PM01	0.00
4	2006-11-10	A	REFIT BUCKET TOOTH	PM01	1157.27

```
vocab=nd.load_vocab('excavators')#.dropna(subset=['alias'])
vocab
```

tokens	NE	alias	notes	score
replace	S	replace	NaN	0.033502
bucket	I	bucket	NaN	0.018969
repair	S	repair	NaN	0.017499
grease	I	grease	NaN	0.017377
leak	P	leak	NaN	0.016591
...	...	...	...	...
1boily 19	NaN	NaN	NaN	0.000046
shd 1fitter	NaN	NaN	NaN	0.000046
19 01	NaN	NaN	NaN	0.000046
01 10	NaN	NaN	NaN	0.000046
1fitter 1boily	NaN	NaN	NaN	0.000046

6767 rows × 4 columns

```
iob = kex.iob_extractor(df.OriginalShorttext, vocab)
iob
```

	token	NE	doc_id
0	bucket	B-I	0
1	won	O	0
2	open	O	0
3	bucket	B-I	1
4	cyl	B-I	1
...	...	...	...
24663	fault	B-P	5484
24664	front	O	5484
24665	found	O	5484
24666	wire	B-I	5484
24667	no	O	5484

24668 rows × 3 columns

NER Example: Using IOB output with NLTK

Much of the code in this exampe is adapted from the following tutorial:

https://towardsdatascience.com/named-entity-recognition-and-classification-with-scikit-learn-f05372f07ba2

```
import numpy as np
import pandas as pd
from sklearn.feature_extraction import DictVectorizer
from sklearn.model_selection import train_test_split
```

```
from nestor import keyword as kex
```

```
import sklearn_crfsuite
from sklearn_crfsuite import scorers
from sklearn_crfsuite import metrics
from collections import Counter
import nestor.datasets as dat
```

Load data

Here, we are loading the excavator dataset and associated vocabulary from the Nestor package.

To use this workflow with your own dataset and Nestor tagging, set up the following dataframes:

```
df = pd.read_csv(
    "original_data.csv"
)

df_1grams = pd.read_csv(
    "vocab1g.csv",
    index_col=0
)

df_ngrams = pd.read_csv(
    "vocabNg.csv",
    index_col=0
)
```

```
df = dat.load_excavators()
# This vocab data inclues 1grams and Ngrams
df_vocab = dat.load_vocab("excavators")
```

Prepare data for modeling

Select column(s) that inlcude text.

```
nlp_select = kex.NLPSelect(columns=['OriginalShorttext'])
raw_text = nlp_select.transform(df.head(100))  # fixme (using abridged dataset here for efficiency)
```

Pass text data and vocab files from Nestor through `iob_extractor`

```
iob = kex.iob_extractor(raw_text, df_vocab)
```

Create X and y data, where y is the IOB labels

```
X = iob.drop('NE', axis=1)
v = DictVectorizer(sparse=False)
X = v.fit_transform(X.to_dict('records'))
y = iob.NE.values
classes = np.unique(y)
classes = classes.tolist()
```

SentenceGetter helper class

```
class SentenceGetter(object):

    def __init__(self, data):
        self.n_sent = 1
        self.data = data
        self.empty = False
        agg_func = lambda s: [(w, t) for w, t in zip(s['token'].values.tolist(),
                                                     s['NE'].values.tolist())]
        self.grouped = self.data.groupby('doc_id').apply(agg_func)
        self.sentences = [s for s in self.grouped]
```

```
    def get_next(self):
        try:
            s = self.grouped['Sentence: {}'.format(self.n_sent)]
            self.n_sent += 1
            return s
        except:
            return None
```

Feature vector helper functions

```
def word2features(sent, i):
    """
        Creates feature vectors, accounting for surrounding tokens and whether or not the token is a number
    """
    word = sent[i][0]

    features = {
        'bias': 1.0,
        'word.lower()': word.lower(),
        'word[-3:]': word[-3:],
        'word[-2:]': word[-2:],
        'word.isdigit()': word.isdigit(),
    }
    if i > 0:
        word1 = sent[i - 1][0]
        features.update({
            '-1:word.isdigit()': word1.isdigit(),
        })
    else:
        features['BOS'] = True
    if i < len(sent) - 1:
        word1 = sent[i + 1][0]
        features.update({
            '+1:word.isdigit()': word1.isdigit(),
        })
    else:
        features['EOS'] = True

    return features
```

```
def sent2features(sent):
    return [word2features(sent, i) for i in range(len(sent))]
```

```
def sent2labels(sent):
    return [label for token, label in sent]
```

```
def sent2tokens(sent):
    return [token for token, label in sent]
```

Prepare data for modeling

```
getter = SentenceGetter(iob)
mwos = getter.sentences
```

```
X = [sent2features(mwo) for mwo in mwos]
y = [sent2labels(mwo) for mwo in mwos]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

Train and test model

This example uses a CFR model; however, this is only one of many ways to perform NER

```
crf = sklearn_crfsuite.CRF(
    algorithm='lbfgs',
    c1=0.1,
    c2=0.1,
    max_iterations=100,
    all_possible_transitions=True
)
```

```
new_classes = classes.copy()
new_classes.pop()
```

```
'O'
```

```
crf.fit(X_train, y_train)
y_pred = crf.predict(X_test)
print(metrics.flat_classification_report(y_test, y_pred, labels = new_classes))
```

```
              precision    recall  f1-score   support

         B-I       0.90      0.72      0.80        61
         B-P       0.57      0.80      0.67         5
        B-PI       0.67      1.00      0.80         2
         B-S       0.80      0.67      0.73        12
        B-SI       0.67      0.91      0.77        11
         I-I       0.33      0.38      0.35         8
        I-PI       0.67      1.00      0.80         2
         I-S       1.00      1.00      1.00         1
        I-SI       0.71      0.91      0.80        11

   micro avg       0.76      0.74      0.75       113
   macro avg       0.70      0.82      0.75       113
weighted avg       0.79      0.74      0.75       113
```

```
/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/sklearn/utils/validation.py:67: FutureWarning: Pass labels=['B-I', 'B-P', 'B-PI', 'B-S', 'B-SI', 'I-I', 'I-PI',
'I-S', 'I-SI'] as keyword args. From version 0.25 passing these as positional arguments will result in an error
  warnings.warn("Pass {} as keyword args. From version 0.25 "
```

NER Example: Using IOB output SpaCy

```
import os
import pandas as pd
from nestor import keyword as kex
import nestor.datasets as dat
from sklearn.model_selection import train_test_split
```

Helper functions

```
def create_iob_format_data(df_iob: pd.DataFrame, ner_file_path: str):
    """
    Create .iob file with token-per-line IOB format
    (see https://github.com/explosion/spaCy/blob/master/extra/example_data/ner_example_data/ner-token-per-line.iob
        for example format)

    Parameters
    ----------
    df_iob: DataFrame created by kex.iob_extractor()
    ner_file_path: pathname for where to save the file in IOB-formatted output, use ".iob" extension

    Returns
    -------

    """
    # to do: make sure that
    # Convert IOB DataFrame to token-per-line tsv file
    df_iob[["token", "NE"]].to_csv(ner_file_path, sep="\t", index=False, header=False)
```

```
def convert_iob_to_spacy_file(ner_file_path: str):
    """

    Parameters
    ----------
    ner_file_path: pathname for where to save the file in IOB-formatted output, use ".iob" extension, must be in format
        as shown here: https://github.com/explosion/spaCy/blob/master/extra/example_data/ner_example_data/ner-token-per-line.iob

    Returns
    -------

    """
    # todo: make this command customizable, handle tokens better (actually need to group by MWO)
    os.system("python -m spacy convert -c ner -s -n 10 -b en_core_web_sm " + ner_file_path + " .")
```

Load data

Here, we are loading the excavator dataset and associated vocabulary from the Nestor package.

To use this workflow with your own dataset and Nestor tagging, set up the following dataframes:

```
df = pd.read_csv(
    "original_data.csv"
)

df_1grams = pd.read_csv(
    "vocab1g.csv",
    index_col=0
)

df_ngrams = pd.read_csv(
    "vocabNg.csv",
    index_col=0
)
```

```
df = dat.load_excavators()
# This vocab data inclues 1grams and Ngrams
df_vocab = dat.load_vocab("excavators")
```

Prepare data for modeling

Select column(s) that inlcude text.

Split data into training and test sets.

```
nlp_select = kex.NLPSelect(columns = ['OriginalShorttext'])
raw_text = nlp_select.transform(df)
train, test = train_test_split(raw_text, test_size=0.2, random_state=1, shuffle=False)
test = test.reset_index(drop=True)
```

Pass text data and vocab files from Nestor through `iob_extractor`

```
iob_train = kex.iob_extractor(train, df_vocab)
iob_test = kex.iob_extractor(test, df_vocab)
```

Create `.iob` files (these are essentially tsv files with proper IOB tag format). Convert `.iob` files to `.spacy` binary files

```
# pathname/document title should match what is in `congif.cfg file`
create_iob_format_data(iob_train, "iob_data.iob")
convert_iob_to_spacy_file("iob_data.iob")
create_iob_format_data(iob_test, "iob_valid.iob")
convert_iob_to_spacy_file("iob_valid.iob")
```

```
Traceback (most recent call last):
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 188, in _run_module_as_main
    mod_name, mod_spec, code = _get_module_details(mod_name, _Error)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 147, in _get_module_details
    return _get_module_details(pkg_main_name, error)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 111, in _get_module_details
    __import__(pkg_name)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/__init__.py", line 15, in <module>
    from .cli.info import info  # noqa: F401
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/cli/__init__.py", line 3, in <module>
    from ._util import app, setup_cli  # noqa: F401
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/cli/_util.py", line 8, in <module>
    import typer
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/__init__.py", line 29, in <module>
    from .main import Typer as Typer
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/main.py", line 11, in <module>
    from .completion import get_completion_inspect_parameters
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/completion.py", line 10, in <module>
    import click._bashcomplete
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
ModuleNotFoundError: No module named 'click._bashcomplete'
Traceback (most recent call last):
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 188, in _run_module_as_main
    mod_name, mod_spec, code = _get_module_details(mod_name, _Error)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 147, in _get_module_details
    return _get_module_details(pkg_main_name, error)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 111, in _get_module_details
    __import__(pkg_name)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/__init__.py", line 15, in <module>
    from .cli.info import info  # noqa: F401
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/cli/__init__.py", line 3, in <module>
    from ._util import app, setup_cli  # noqa: F401
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/cli/_util.py", line 8, in <module>
    import typer
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/__init__.py", line 29, in <module>
    from .main import Typer as Typer
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/main.py", line 11, in <module>
    from .completion import get_completion_inspect_parameters
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/completion.py", line 10, in <module>
    import click._bashcomplete
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
ModuleNotFoundError: No module named 'click._bashcomplete'
```

SpaCy model

Run data through basic spaCy training (relies on `spacy_config.cfg`). This stage can be customized as needed for your particular modeling and analysis.

```
# Run data through basic spacy training for proof of concept.
os.system("python -m spacy train spacy_config.cfg --output ./output")
```

```
Traceback (most recent call last):
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 188, in _run_module_as_main
    mod_name, mod_spec, code = _get_module_details(mod_name, _Error)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 147, in _get_module_details
    return _get_module_details(pkg_main_name, error)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/runpy.py", line 111, in _get_module_details
    __import__(pkg_name)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/__init__.py", line 15, in <module>
    from .cli.info import info  # noqa: F401
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/cli/__init__.py", line 3, in <module>
    from ._util import app, setup_cli  # noqa: F401
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/spacy/cli/_util.py", line 8, in <module>
    import typer
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/__init__.py", line 29, in <module>
    from .main import Typer as Typer
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/main.py", line 11, in <module>
    from .completion import get_completion_inspect_parameters
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/typer/completion.py", line 10, in <module>
    import click._bashcomplete
  File "/home/tbsexton/miniconda3/envs/nestor-docs/lib/python3.9/site-packages/poetry_dynamic_versioning/__init__.py", line 416, in alt_import
    module = _state.original_import_func(name, globals, locals, fromlist, level)
ModuleNotFoundError: No module named 'click._bashcomplete'
```

```
256
```

6. API Reference

6.1 `nestor.settings`

`NestorParams`

Temporary subclass of `dict` to manage nestor contexts.

> To be re-factored as typed dataclasses.
>
> TODO: allow context-based switching (a.k.a matplotlib xParams style) A valid nestor config `yaml` is formated with these feilds:

```yaml
entities:
  types:
    atomic:
        code: description
        ...
    derived:
        ...
    hole:
        ...
  rules:
    code:
      - [codeA,codeB]
      ...
    ...

datatypes:
    ...
```

For the default `nestor.CFG`, we provide a schema based on nestor's roots in manufacturing maintenance:

```yaml
token_patt = '(?u)    \w\w+    '
entities:
  types:
    'atomic': # atomic types
      'P': 'Problem'
      'I': 'Item'
      'S': 'Solution'
    'derived':  # only made from atoms
      'PI': 'Object Fault'
      'SI': 'Object Resolution'
    'hole':
      'U': 'Unknown'
      'X': 'Non Entity'
      # 'NA': 'Not Annotated'

  rules:
    # two items makes one new item
    'I':
      - ['I','I']
    'PI':
      - ['P','I']
    'SI':
      - ['S','I']
    # redundancies
    'X':
      - ['P', 'P']
      - ['S', 'S']
      - ['P', 'S']
  # note: could try ordered as 'X':{1:'P',2:'S'}, etc.

datatypes:
  issue:
    description:
      problem: 'Description of Problem'
      solution: 'Description of Solution'
      cause: 'Description of Cause'
      effect: 'Description of Observed Symptoms (Effects)'
    machine_down: 'Machine Up/Down'
    necessary_part: 'Necessary Part'
    part_in_process: 'Part in Process'
    cost: 'Maintenance Cost'
    id: 'MWO ID Number'
    date:
      machine_down: 'Machine Down Time-stamp'
      workorder_start: 'Work Order Start Time-stamp'
      maintenance_technician_arrive: 'Maintenance Technician Arrives Time-stamp'
      solution_found: 'Problem Found Time-stamp'
      part_ordered: 'Part(s) Ordered Time-stamp'
```

```
        part_received: 'Part(s) Received Time-stamp'
        solution_solve: 'Problem Solved Time-stamp'
        machine_up: 'Machine Up Time-stamp'
        workorder_completion: 'Work Order Completion Time-stamp'

    technician:
      name: 'Maintenance Technician'
      skills: 'Skill(s)'
      crafts: 'Craft(s)'

    operator:
      name: 'Operator'

    machine:
      name: 'Asset ID'
      manufacturer: 'Original Equipment Manufacturer'
      type: 'Machine Type'

    location:
      name: "Location"
```

While future releases are focused on bringing more flexibility to users to define their own types, it is still possible to use these settings for a wide variety of tasks.

datatype_search(self, property_name)

find any datatype that has a specific key

Source code in `nestor/settings.py`

```python
def datatype_search(self, property_name):
    """find any datatype that has a specific key"""
    return find_path_from_key(self["datatypes"], property_name)
```

`nestor_params()`

Function to instantiate a :class: `nestor.NestorParams` instance from the default nestor config/type .yaml files

For now, provides the default `settings.yaml`, based on maintenance work-orders.

Returns:

Type	Description
`nestor.NestorParams`	context-setting config object for other nestor behavior

Source code in `nestor/settings.py`

```python
def nestor_params():
    """Function to instantiate a :class:`nestor.NestorParams` instance from
    the default nestor config/type .yaml files

    For now, provides the default `settings.yaml`, based on maintenance work-orders.

    Returns:
        nestor.NestorParams: context-setting config object for other nestor behavior
    """
    fnames = nestor_fnames()
    # could check they exist, probably
    return nestor_params_from_files(fnames)
```

`nestor_params_from_files(fname)`

Build up a `nestor.NestorParams` object from a passed config file locations

Parameters:

Name	Type	Description	Default
`fname`	`pathlib.Path`	location of a valid `.yaml` that defines a NestorParams object	*required*

Returns:

Type	Description
`nestor.NestorParams`	context-setting config object for other nestor behavior

Source code in `nestor/settings.py`

```
def nestor_params_from_files(fname):
    """
    Build up a `nestor.NestorParams` object from a passed config file locations

    Args:
        fname (pathlib.Path): location of a valid `.yaml` that defines a NestorParams object

    Returns:
        nestor.NestorParams: context-setting config object for other nestor behavior
    """

    settings_dict = yaml.safe_load(open(fname))
    cfg = NestorParams(**settings_dict)
    return cfg
```

6.2 `nestor.keyword`

`NLPSelect`

Extract specified natural language columns

Starting from a pd.DataFrame, combine `columns` into a single series containing lowercased text with punctuation and excess newlines removed. Using the `special_replace` dict allows for arbitrary mapping during the cleaning process, for e.g. a priori normalization.

Parameters:

Name	Type	Description	Default
`columns(int,list`	`of int,str`	names/positions of data columns to extract, clean, and merge	*required*
`special_replace(dict,None)`		mapping from strings to normalized strings (known a priori)	*required*
`together(pd.Series)`		merged text, before any cleaning/normalization	*required*
`clean_together(pd.Series)`		merged text, after cleaning (output of `transform`)	*required*

`get_params(self, deep=True)`

Retrieve parameters of the transformer for sklearn compatibility.

Parameters:

Name	Type	Description	Default
`deep`		(Default value = True)	`True`

Source code in `nestor/keyword.py`

```
def get_params(self, deep=True):
    """Retrieve parameters of the transformer for sklearn compatibility.

    Args:
      deep:  (Default value = True)

    Returns:

    """
    return dict(
        columns=self.columns, names=self.names, special_replace=self.special_replace
    )
```

`transform(self, X, y=None)`

get clean column of text from column(s) of raw text in a dataset

Depending on which of Union[List[Union[int,str]],int,str] `self.columns` is, this will extract desired columns (of text) from positions, names, etc. in the original dataset `X`.

Columns will be merged, lowercased, and have punctuation and hanging newlines removed.

Parameters:

Name	Type	Description	Default
`X(pandas.DataFrome)`		dataset containing certain columns with natural language text.	*required*
`y(None,`	`optional`	(Default value = None)	*required*

Returns:

Type	Description
`clean_together(pd.Series)`	a single column of merged, cleaned text

Source code in `nestor/keyword.py`

```
def transform(self, X, y=None):
    """get clean column of text from column(s) of raw text in a dataset

    Depending on which of Union[List[Union[int,str]],int,str]
    `self.columns` is, this will extract desired columns (of text) from
    positions, names, etc. in the original dataset `X`.

    Columns will be merged, lowercased, and have punctuation and hanging
    newlines removed.

    Args:
      X(pandas.DataFrome): dataset containing certain columns with natural language text.
      y(None, optional):  (Default value = None)

    Returns:
       clean_together(pd.Series): a single column of merged, cleaned text

    """
    if isinstance(self.columns, list):  # user passed a list of column labels
        if all([isinstance(x, int) for x in self.columns]):
            nlp_cols = list(
                X.columns[self.columns]
            )  # select columns by user-input indices
        elif all([isinstance(x, str) for x in self.columns]):
            nlp_cols = self.columns  # select columns by user-input names
        else:
            print("Select error: mixed or wrong column type.")  # can't do both
            raise Exception
    elif isinstance(self.columns, int):  # take in a single index
        nlp_cols = [X.columns[self.columns]]
    else:
        nlp_cols = [self.columns]  # allow...duck-typing I guess? Don't remember.

    def _robust_cat(df, cols):
        """pandas doesn't like batch-cat of string cols...needs 1st col

        Args:
          df:
          cols:

        Returns:

        """
        if len(cols) <= 1:
            return df[cols].astype(str).fillna("").iloc[:, 0]
        else:
            return (
                df[cols[0]]
                .astype(str)
                .str.cat(df.loc[:, cols[1:]].astype(str), sep=" ", na_rep="",)
            )

    def _clean_text(s, special_replace=None):
        """lower, rm newlines and punct, and optionally special words

        Args:
          s:
          special_replace:  (Default value = None)

        Returns:

        """
        raw_text = (
            s.str.lower()  # all lowercase
            .str.replace("\n", " ")  # no hanging newlines
            .str.replace("[{}]".format(string.punctuation), " ")
        )
        if special_replace is not None:
            rx = re.compile("|".join(map(re.escape, special_replace)))
            # allow user-input special replacements.
            return raw_text.str.replace(
                rx, lambda match: self.special_replace[match.group(0)]
            )
        else:
            return raw_text

    self.together = X.pipe(_robust_cat, nlp_cols)
    self.clean_together = self.together.pipe(
        _clean_text, special_replace=self.special_replace
    )
    return self.clean_together
```

`TagExtractor`

Wrapper for TokenExtractor to apply a *Nestor* thesaurus or vocabulary definition on-top of the token extraction process. Also provides several useful methods as a result.

`__init__(self, thesaurus=None, group_untagged=True, filter_types=None, verbose=False, output_type=<TagRep.binary: 'binary'>, **tfidf_kwargs)` special

Identical to the TokenExtractor initialization, Except for the addition of an optional `vocab` argument that allows for pre-defined thesaurus/dictionary mappings of tokens to named entities (see generate_vocabulary_df) to get used in the transformation doc-term form.

Rather than outputting a TF-IDF-weighted sparse matrix, this transformer outputs a Multi-column `pd.DataFrame` with the top-level columns being current tag-types in `nestor.CFG`, and the sub-level being the actual tokens/compound-tokens.

Source code in `nestor/keyword.py`

```python
def __init__(
    self,
    thesaurus=None,
    group_untagged=True,
    filter_types=None,
    verbose=False,
    output_type: TagRep = TagRep["binary"],
    **tfidf_kwargs,
):
    """
    Identical to the [TokenExtractor](nestor.keyword.TokenExtractor) initialization,
    Except for the addition of an optional `vocab` argument that allows for pre-defined
    thesaurus/dictionary mappings of tokens to named entities
    (see [generate_vocabulary_df](nestor.keyword.generate_vocabulary_df))
    to get used in the transformation doc-term form.

    Rather than outputting a TF-IDF-weighted sparse matrix, this transformer outputs a Multi-column
    `pd.DataFrame` with the top-level columns being current tag-types in `nestor.CFG`, and the sub-level
    being the actual tokens/compound-tokens.

    """
    # super().__init__()
    default_kws = dict(
        input="content",
        ngram_range=(1, 1),
        stop_words="english",
        sublinear_tf=True,
        smooth_idf=False,
        max_features=5000,
        token_pattern=nestorParams.token_pattern,
    )
    default_kws.update(**tfidf_kwargs)

    super().__init__(**default_kws)  # get internal attrs from parent
    self._tokenmodel = TokenExtractor(
        **default_kws
    )  # persist an instance for composition

    self.group_untagged = group_untagged
    self.filter_types = filter_types
    self.output_type = output_type
    self._verbose = verbose
    self._thesaurus = thesaurus
    self.tfidf_ = None

    self.tag_df_ = None
    self.iob_rep_ = None
    self.multi_rep_ = None

    self.tag_completeness_ = None
    self.num_complete_docs_ = None
    self.num_empty_docs_ = None
```

`fit(self, documents, y=None)`

Learn a vocabulary dictionary of tokens in raw documents.

Parameters:

Name	Type	Description	Default
`documents`	`pd.Series, Iterable`	Iterable of raw documents	*required*
`y`		(Default value = None)	`None`

Returns:

Type	Description
	self

Source code in `nestor/keyword.py`

```
def fit(self, documents, y=None):
    # self._tokenmodel.fit(documents)
    self.tfidf_ = self._tokenmodel.fit_transform(documents)
    # check_is_fitted(self._tokenmodel, msg="The tfidf vector is not fitted")
    tag_df = tag_extractor(
        self._tokenmodel,
        documents,
        vocab_df=self.thesaurus,
        group_untagged=self.group_untagged,
    )
    if self.filter_types:
        tag_df = pick_tag_types(tag_df, self.filter_types)

    self.tag_df = tag_df
    self.tags_as_iob = documents
    self.tags_as_lists = tag_df
    self.set_stats()
    if self._verbose:
        self.report_completeness()
    return self
```

`fit_transform(self, documents, y=None)`

Turn TokenExtractor instances and raw-text into binary occurrences.

Wrapper for the TokenExtractor to streamline the generation of tags from text. Determines the documents in `raw_text` that contain each of the tags in `vocab_df`, using a TokenExtractor self object (i.e. the tfidf vocabulary).

As implemented, this function expects an existing self object, though in the future this may be changed to a class-like functionality (e.g. sklearn's AdaBoostClassifier, etc) which wraps a self into a new one.

Parameters:

Name	Type	Description	Default
`self`	`object KeywordExtractor`	instantiated, can be pre-trained	*required*
`raw_text`	`pd.Series`	contains jargon/slang-filled raw text to be tagged	*required*
`vocab_df`	`pd.DataFrame`	An existing vocabulary dataframe or .csv filename, expected in the format of kex.generate_vocabulary_df(). (Default value = None)	*required*
`readable`	`bool`	whether to return readable, categorized, comma-sep str format (takes longer) (Default value = False)	*required*
`group_untagged`	`bool`	whether to group untagged tokens into a catch-all "_untagged" tag	*required*

Returns:

Type	Description
`pd.DataFrame`	extracted tags for each document, whether binary indicator (default) or in readable, categorized, comma-sep str format (readable=True, takes longer)

Source code in `nestor/keyword.py`

```
@documented_at(tag_extractor, transformer="self")
def fit_transform(self, documents, y=None):
    """Fit transformer on `documents` and return the binary, hierarchical """
    self.fit(documents)

    return self.transform(documents)
```

`transform(self, documents, y=None)`

Source code in `nestor/keyword.py`

```
def transform(self, documents, y=None):
    """
    """
    check_is_fitted(self, "tag_df_")

    if self.output_type == TagRep.multilabel:
        return self.tags_as_lists
    elif self.output_type == TagRep.iob:
        return self.tags_as_iob
    else:
        return self.tag_df
```

`TagRep`

available representation of tags in documents

`TokenExtractor`

A wrapper for the sklearn TfidfVectorizer class, with utilities for ranking by total tf-idf score, and getting a list of vocabulary.

Valid options are given below from sklearn docs.

`ranks_` property writable

Retrieve the rank of each token, for sorting. Uses summed scoring over the TF-IDF for each token, so that: S_t = \Sum_{d
ext{TF-IDF}_t

`scores_` property writable

Returns actual scores of tokens, for progress-tracking (min-max-normalized)

Returns:

Type	Description
`numpy.array`	

`sumtfidf_` property writable

sum of the tf-idf scores for each token over all documents.

Thought to approximate mutual information content of a given string.

`vocab_` property writable

ordered list of tokens, rank-ordered by summed-tf-idf (see :func: `~nestor.keyword.TokenExtractor.ranks_`)

`__init__(self, input='content', ngram_range=(1, 1), stop_words='english', sublinear_tf=True, smooth_idf=False, max_features=5000, token_pattern='\\b\\w\\w+\\b', **tfidf_kwargs)` special

Initialize the extractor

Parameters:

Name	Type	Description	Default
`input`	`string`	{'filename', 'file', 'content'} If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze. If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory. Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.	`'content'`
`ngram_range`	`tuple`	(min_n, max_n), default=(1,1) The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.	`(1, 1)`
`stop_words`	`string`	{'english'} (default), list, or None If a string, it is passed to _check_stop_list and the appropriate stop list is returned. 'english' is currently the only supported string value. If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`. If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.	`'english'`
`max_features`	`int or None`	If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus. This parameter is ignored if vocabulary is not None. (default=5000)	`5000`
`smooth_idf`	`boolean`	Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions. (default=False)	`False`
`sublinear_tf`	`boolean`	(Default value = True) Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).	`True`
`**tfidf_kwargs`		other arguments passed to `sklearn.TfidfVectorizer`	`{}`

Source code in `nestor/keyword.py`

```
def __init__(
    self,
    input="content",
    ngram_range=(1, 1),
    stop_words="english",
    sublinear_tf=True,
    smooth_idf=False,
    max_features=5000,
    token_pattern=nestorParams.token_pattern,
    **tfidf_kwargs,
):
    """Initialize the extractor

    Args:
       input (string): {'filename', 'file', 'content'}
            If 'filename', the sequence passed as an argument to fit is
            expected to be a list of filenames that need reading to fetch
            the raw content to analyze.

            If 'file', the sequence items must have a 'read' method (file-like
            object) that is called to fetch the bytes in memory.
            Otherwise the input is expected to be the sequence strings or
            bytes items are expected to be analyzed directly.
       ngram_range (tuple): (min_n, max_n), default=(1,1)
            The lower and upper boundary of the range of n-values for different
            n-grams to be extracted. All values of n such that min_n <= n <= max_n
            will be used.
       stop_words (string): {'english'} (default), list, or None
            If a string, it is passed to _check_stop_list and the appropriate stop
            list is returned. 'english' is currently the only supported string
            value.

            If a list, that list is assumed to contain stop words, all of which
            will be removed from the resulting tokens.
            Only applies if ``analyzer == 'word'``.

            If None, no stop words will be used. max_df can be set to a value
            in the range [0.7, 1.0) to automatically detect and filter stop
            words based on intra corpus document frequency of terms.
       max_features (int or None):
            If not None, build a vocabulary that only consider the top
            max_features ordered by term frequency across the corpus.
            This parameter is ignored if vocabulary is not None.
            (default=5000)
       smooth_idf (boolean):
            Smooth idf weights by adding one to document frequencies, as if an
            extra document was seen containing every term in the collection
            exactly once. Prevents zero divisions. (default=False)
       sublinear_tf (boolean): (Default value = True)
            Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

       **tfidf_kwargs: other arguments passed to `sklearn.TfidfVectorizer`
    """
    self.default_kws = dict(
        {
            "input": input,
            "ngram_range": ngram_range,
            "stop_words": stop_words,
            "sublinear_tf": sublinear_tf,
            "smooth_idf": smooth_idf,
            "max_features": max_features,
            "token_pattern": token_pattern,
        }
    )

    self.default_kws.update(tfidf_kwargs)
    self._model = TfidfVectorizer(**self.default_kws)
    self._tf_tot = None

    self._ranks = None
    self._vocab = None
    self._scores = None
```

`fit(self, documents, y=None)`

Learn a vocabulary dictionary of tokens in raw documents.

Parameters:

Name	Type	Description	Default
`documents`	`pd.Series, Iterable`	Iterable of raw documents	*required*
`y`		(Default value = None)	`None`

Returns:

Type	Description
	self

Source code in `nestor/keyword.py`

```
def fit(self, documents, y=None):
    """
    Learn a vocabulary dictionary of tokens in raw documents.
    Args:
      documents (pd.Series, Iterable): Iterable of raw documents
      y:  (Default value = None)

    Returns:
      self
    """
    _ = self.fit_transform(documents)
    return self
```

`fit_transform(self, documents, y=None, **fit_params)`

transform a container of text documents to TF-IDF Sparse Matrix

Parameters:

Name	Type	Description	Default
`documents`	`pd.Series, Iterable`	Iterable of raw documents	*required*
`y`		(Default value = None) unused	`None`
`**fit_params`		kwargs passed to underlying TfidfVectorizer	`{}`

Returns:

Type	Description
`X_tf`	array of shape (n_samples, n_features) document-term matrix

Source code in `nestor/keyword.py`

```
def fit_transform(self, documents, y=None, **fit_params):
    """transform a container of text documents to TF-IDF Sparse Matrix

    Args:
      documents (pd.Series, Iterable): Iterable of raw documents
      y:  (Default value = None) unused
      **fit_params: kwargs passed to underlying TfidfVectorizer

    Returns:
      X_tf: array of shape (n_samples, n_features)
            document-term matrix

    """
    if isinstance(documents, pd.Series):
        documents = _series_itervals(documents)
    if y is None:
        X_tf = self._model.fit_transform(documents)
    else:
        X_tf = self._model.fit_transform(documents, y)
    self.sumtfidf_ = X_tf.sum(axis=0)

    ranks = self.sumtfidf_.argsort()[::-1]
    if len(ranks) > self.default_kws["max_features"]:
        ranks = ranks[: self.default_kws["max_features"]]
    self.ranks_ = ranks

    self.vocab_ = np.array(self._model.get_feature_names())[self.ranks_]
    scores = self.sumtfidf_[self.ranks_]
    self.scores_ = (scores - scores.min()) / (scores.max() - scores.min())
    return X_tf
```

`thesaurus_template(self, filename=None, init=None)`

make correctly formatted entity vocabulary (token->tag+type)

Helper method to create a formatted pandas.DataFrame and/or a .csv containing the token--tag/alias--classification relationship. Formatted as jargon/slang tokens, the Named Entity classifications, preferred labels, notes, and tf-idf summed scores:

tokens	NE	alias	notes	scores
myexample	I	example	"e.g"	0.42

This is intended to be filled out in excel or using the Tagging Tool UI

- `nestor-qt`
- `nestor-web`

Parameters:

Name	Type	Description	Default
`self`	`TokenExtractor`	the (TRAINED) token extractor used to generate the ranked list of vocab.	*required*
`init`	`str or pd.Dataframe`	file location of csv or dataframe of existing vocab list to read and update token classification values from	`None`

Returns:

Type	Description
`pd.Dataframe`	the correctly formatted vocabulary list for token:NE, alias matching

Source code in `nestor/keyword.py`

```
@documented_at(generate_vocabulary_df, transformer="self")
def thesaurus_template(self, filename=None, init=None):
    return generate_vocabulary_df(self, filename=filename, init=init)
```

`transform(self, documents)`

transform documents into document-term matrix

Parameters:

Name	Type	Description	Default
`documents`			*required*

Returns:

Type	Description
`X_tf`	array of shape (n_samples, n_features) document-term matrix

Source code in `nestor/keyword.py`

```
def transform(self, documents):
    """transform documents into document-term matrix

    Args:
      documents:

    Returns:
      X_tf: array of shape (n_samples, n_features)
            document-term matrix


    """

    check_is_fitted(self._model, msg="The tfidf vector is not fitted")

    if isinstance(documents, pd.Series):
        X = _series_itervals(documents)
    X_tf = self._model.transform(X)
    self.sumtfidf_ = X_tf.sum(axis=0)
    return X_tf
```

`generate_vocabulary_df(transformer, filename=None, init=None)`

make correctly formatted entity vocabulary (token->tag+type)

Helper method to create a formatted pandas.DataFrame and/or a .csv containing the token--tag/alias--classification relationship. Formatted as jargon/slang tokens, the Named Entity classifications, preferred labels, notes, and tf-idf summed scores:

tokens	NE	alias	notes	scores
myexample	I	example	"e.g"	0.42

This is intended to be filled out in excel or using the Tagging Tool UI

- `nestor-qt`
- `nestor-web`

Parameters:

Name	Type	Description	Default
`transformer`	`TokenExtractor`	the (TRAINED) token extractor used to generate the ranked list of vocab.	*required*
`init`	`Union[str, pandas.core.frame.DataFrame]`	file location of csv or dataframe of existing vocab list to read and update token classification values from	`None`

Returns:

Type	Description
`pd.Dataframe`	the correctly formatted vocabulary list for token:NE, alias matching

Source code in `nestor/keyword.py`

```
def generate_vocabulary_df(
    transformer, filename=None, init: Union[str, pd.DataFrame] = None
):
    """ make correctly formatted entity vocabulary (token->tag+type)

    Helper method to create a formatted pandas.DataFrame and/or a .csv containing
    the token--tag/alias--classification relationship. Formatted as jargon/slang tokens,
    the Named Entity classifications, preferred labels, notes, and tf-idf summed scores:

    tokens | NE | alias | notes | scores
    --- | --- | --- | --- | ---
    myexample| I | example | "e.g"| 0.42

    This is intended to be filled out in excel or using the Tagging Tool UI

    - [`nestor-qt`](https://github.com/usnistgov/nestor-qt)
    - [`nestor-web`](https://github.com/usnistgov/nestor-web)

    Parameters:
        transformer (TokenExtractor): the (TRAINED) token extractor used to generate the ranked list of vocab.
        filename (str, optional) the file location to read/write a csv containing a formatted vocabulary list
        init (str or pd.Dataframe, optional): file location of csv or dataframe of existing vocab list to read and update
            token classification values from

    Returns:
        pd.Dataframe: the correctly formatted vocabulary list for token:NE, alias matching
    """

    try:
        check_is_fitted(
            transformer._model, "vocabulary_", msg="The tfidf vector is not fitted"
        )
    except NotFittedError:
        if (filename is not None) and Path(filename).is_file():
            print("No model fitted, but file already exists. Importing...")
            return pd.read_csv(filename, index_col=0)
        elif (init is not None) and Path(init).is_file():
            print("No model fitted, but file already exists. Importing...")
            return pd.read_csv(init, index_col=0)
        else:
            raise

    df = (
        pd.DataFrame(
            {
                "tokens": transformer.vocab_,
                "NE": "",
                "alias": "",
                "notes": "",
                "score": transformer.scores_,
            }
        )
        # .loc[:,["tokens", "NE", "alias", "notes", "score"]]
        .pipe(lambda df: df[~df.tokens.duplicated(keep="first")]).set_index("tokens")
    )

    if init is None:
        if (filename is not None) and Path(filename).is_file():
            init = filename
            print("attempting to initialize with pre-existing vocab")

    if init is not None:
        df.NE = np.nan
        df.alias = np.nan
        df.notes = np.nan
        if isinstance(init, Path) and init.is_file():  # filename is passed
            df_import = pd.read_csv(init, index_col=0)
        else:
            try:  # assume input pandas df
                df_import = init.copy()
            except AttributeError:
                print("File not Found! Can't import!")
                raise
        df.update(df_import)
        # print('intialized successfully!')
        df.fillna("", inplace=True)

    if filename is not None:
        df.to_csv(filename)
        print("saved locally!")
    return df
```

`get_multilabel_representation(tag_df)`

Turn binary tag occurrences into strings of comma-separated tags

Given a hierarchical column-set of (entity-types, tag), where each row is a document and the binary-valued elements indicate occurrence (see `nestor.tag_extractor`), use this to get something a little more human-readable. Columns will be entity-types, with elements as comma-separated strings of tags.

Uses some hacks, since categorical from strings tends to assume single (not multi-label) categories per-document. Likely to be re-factored in the future, but used for the `readable=True` flag in `tag_extractor`.

Parameters:

Name	Type	Description	Default
`tag_df`	`pd.DataFrame`	binary occurrence matrix from `tag_extractor`	*required*

Returns:

Type	Description
`pd.DataFrame`	document matrix with columns of tag-types, elements of comma-separated tags of that type.

Source code in `nestor/keyword.py`

```
def get_multilabel_representation(tag_df):
    """Turn binary tag occurrences into strings of comma-separated tags

    Given a hierarchical column-set of (entity-types, tag), where each row is
    a document and the binary-valued elements indicate occurrence
    (see `nestor.tag_extractor`), use this to get something a little more
    human-readable. Columns will be entity-types, with elements as
    comma-separated strings of tags.

    Uses some hacks, since categorical from strings tends to assume single (not
    multi-label) categories per-document. Likely to be re-factored in the future,
    but used for the `readable=True` flag in `tag_extractor`.

    Args:
      tag_df (pd.DataFrame): binary occurrence matrix from `tag_extractor`

    Returns:
       pd.DataFrame: document matrix with columns of tag-types, elements of
       comma-separated tags of that type.

    """
    return _get_readable_tag_df(tag_df)
```

`get_tag_completeness(tag_df, verbose=True)`

completeness, emptiness, and histograms in-between

It's hard to estimate "how good of a job you've done" at annotating your data. One way is to calculate the fraction of documents where all tokens have been mapped to their normalized form (a tag). Conversely, the fraction that have no tokens normalized, at all.

Interpolating between those extremes, we can think of the Positive Predictive Value (PPV, also known as Precision) of our annotations: of the tokens/concepts not cleaned out (ostensibly, the *relevant* ones, how many have been retrieved (i.e. mapped to a known tag)?

Parameters:

Name	Type	Description	Default
`tag_df`	`pd.DataFrame`	hierarchical-column df containing	*required*

Returns:

Type	Description
`tuple`	tuple containing: `tag_pct(pd.Series): PPV/precision for all documents, useful for e.g. histograms` `tag_comp(float): Fraction of documents that are *completely* tagged` `tag_empt(float): Fraction of documents that are completely *untagged*`

Source code in `nestor/keyword.py`

```python
def get_tag_completeness(tag_df, verbose=True):
    """completeness, emptiness, and histograms in-between

    It's hard to estimate "how good of a job you've done" at annotating your
    data. One way is to calculate the fraction of documents where all tokens
    have been mapped to their normalized form (a tag). Conversely, the fraction
    that have no tokens normalized, at all.

    Interpolating between those extremes, we can think of the Positive
    Predictive Value (PPV, also known as Precision) of our annotations: of the
    tokens/concepts not cleaned out (ostensibly, the *relevant* ones, how many
    have been retrieved (i.e. mapped to a known tag)?

    Args:
      tag_df (pd.DataFrame): hierarchical-column df containing

    Returns:
       tuple: tuple containing:

           tag_pct(pd.Series): PPV/precision for all documents, useful for e.g. histograms
           tag_comp(float): Fraction of documents that are *completely* tagged
           tag_empt(float): Fraction of documents that are completely *untagged*

    """

    all_empt = np.zeros_like(tag_df.index.values.reshape(-1, 1))
    tag_pct = 1 - (
        tag_df.get(["NA", "U"], all_empt).sum(axis=1) / tag_df.sum(axis=1)
    )  # TODO: if they tag everything?

    tag_comp = (tag_df.get("NA", all_empt).sum(axis=1) == 0).sum()

    tag_empt = (
        (tag_df.get("I", all_empt).sum(axis=1) == 0)
        & (tag_df.get("P", all_empt).sum(axis=1) == 0)
        & (tag_df.get("S", all_empt).sum(axis=1) == 0)
    ).sum()

    def _report_completeness():
        print(f"Complete Docs: {tag_comp}, or {tag_comp / len(tag_df):.2%}")
        print(f"Tag completeness: {tag_pct.mean():.2f} +/- {tag_pct.std():.2f}")
        print(f"Empty Docs: {tag_empt}, or {tag_empt / len(tag_df):.2%}")

    if verbose:
        _report_completeness()
    return tag_pct, tag_comp, tag_empt
```

`iob_extractor(raw_text, vocab_df_1grams, vocab_df_ngrams=None)`

Use Nestor named entity tags to create IOB format output for NER tasks

This function provides IOB-formatted tagged text, which allows for further NLP analysis. In the output, each token is listed sequentially, as they appear in the raw text. Inside and Beginning Tokens are labeled with "I-" or "B-" and their Named Entity tags; any multi-token entities all receive the same label. Untagged tokens are labeled as "O" (Outside).

Example output (in this example, "PI" is "Problem Item":

token | NE | doc_id an | O | 0 oil | B-PI | 0 leak | I-PI | 0

Parameters:

Name	Type	Description	Default
`raw_text`	`pd.Series`	contains jargon/slang-filled raw text to be tagged	*required*
`vocab_df_1grams`	`pd.DataFrame`	An existing vocabulary dataframe or .csv filename, expected in the format of kex.generate_vocabulary_df(), containing tagged 1-gram tokens vocab_df_ngrams (pd.DataFrame, optional): An existing vocabulary dataframe or .csv filename, expected in the format of kex.generate_vocabulary_df(), containing tagged n-gram tokens (Default value = None)	*required*

Returns:

Type	Description
`pd.DataFrame`	contains row for each token ("token", "NE" (IOB format tag), and "doc_id")

PARAMETERS

raw_text vocab_df_1grams vocab_df_ngrams

Source code in `nestor/keyword.py`

```python
def iob_extractor(raw_text, vocab_df_1grams, vocab_df_ngrams=None):
    """Use Nestor named entity tags to create IOB format output for NER tasks

    This function provides IOB-formatted tagged text, which allows for further NLP analysis. In the output,
    each token is listed sequentially, as they appear in the raw text. Inside and Beginning Tokens are labeled with
    "I-" or "B-" and their Named Entity tags; any multi-token entities all receive the same label.
    Untagged tokens are labeled as "O" (Outside).

    Example output (in this example, "PI" is "Problem Item":

    token | NE | doc_id
    an | O | 0
    oil | B-PI | 0
    leak | I-PI | 0

    Args:
       raw_text (pd.Series): contains jargon/slang-filled raw text to be tagged
       vocab_df_1grams (pd.DataFrame): An existing vocabulary dataframe or .csv filename, expected in the format of
           kex.generate_vocabulary_df(), containing tagged 1-gram tokens
        vocab_df_ngrams (pd.DataFrame, optional): An existing vocabulary dataframe or .csv filename, expected in
        the format of kex.generate_vocabulary_df(), containing tagged n-gram tokens (Default value = None)

    Returns:
        pd.DataFrame: contains row for each token ("token", "NE" (IOB format tag), and "doc_id")

    Parameters
    ----------
    raw_text
    vocab_df_1grams
    vocab_df_ngrams
    """

    # Create IOB output DataFrame
    # iob = pd.DataFrame(columns=["token", "NE", "doc_id"])

    if vocab_df_ngrams is not None:
        # Concatenate 1gram and ngram dataframes
        vocab_df = pd.concat([vocab_df_1grams, vocab_df_ngrams])
        # Get aliased text using ngrams
        # raw_text = token_to_alias(raw_text, vocab_df_ngrams)
    else:
        # Only use 1gram vocabulary provided
        vocab_df = vocab_df_1grams.copy()
        # Get aliased text
        # raw_text = token_to_alias(raw_text, vocab_df_1grams)
        #
    vocab_thesaurus = vocab_df.alias.dropna().to_dict()
    NE_thesaurus = vocab_df.NE.fillna("U").to_dict()

    rx_vocab = regex_match_vocab(vocab_thesaurus, tokenize=True)
    # rx_NE = regex_match_vocab(NE_thesaurus)
    #
    def beginning_token(df: pd.DataFrame) -> pd.DataFrame:
        """after tokens are split and iob column exists"""
        b_locs = df.groupby("token_id", as_index=False).nth(0).index
        df.loc[df.index[b_locs], "iob"] = "B"
        return df

    def outside_token(df: pd.DataFrame) -> pd.DataFrame:
        """after tokens are split and iob,NE columns exist"""
        is_out = df["NE"].isin(nestorParams.holes)
        return df.assign(iob=df["iob"].mask(is_out, "O"))

    tidy_tokens = (  # unpivot the text into one-known-token-per-row
        raw_text.rename("text")
        .rename_axis("doc_id")
        .str.lower()
        .str.findall(rx_vocab)
        # longer series, one-row-per-token
        .explode()
        # it's a dataframe now, with doc_id column
        .reset_index()
        # map tokens to NE, _fast tho_
        .assign(NE=lambda df: regex_thesaurus_normalizer(NE_thesaurus, df.text))
        # regex replace doesnt like nan, so we find the non-vocab tokens and make them unknown
        .assign(NE=lambda df: df.NE.where(df.NE.isin(NE_thesaurus.values()), "U"))
        # now split on spaces and underscores (nestor's compound tokens)
        .assign(token=lambda df: df.text.str.split(r"[_\s]"))
        .rename_axis("token_id")  # keep track of which nestor token was used
        .explode("token")
        .reset_index()
        .assign(iob="I")
        .pipe(beginning_token)
        .pipe(outside_token)
    )
    iob = (
        tidy_tokens.loc[:, ["token", "NE", "doc_id"]]
        .assign(
            NE=tidy_tokens["NE"].mask(tidy_tokens["iob"] == "O", np.nan)
        )  # remove unused NE's
        .assign(
            NE=lambda df: tidy_tokens["iob"]
            .str.cat(df["NE"], sep="-", na_rep="")
            .str.strip("-")
        )  # concat iob-NE
```

```
    )
    return iob
```

`ngram_automatch(voc1, voc2)`

auto-match tag combinations using `nestorParams.entity_rules_map`

Experimental method to auto-match tag combinations into higher-level concepts, primarily to suggest compound entity types to a user.

Used in `nestor.ui`

Parameters:

Name	Type	Description	Default
`voc1`	`pd.DataFrame`	known 1-gram token->tag mapping, with types	*required*
`voc2`	`pd.DataFrame`	current 2-gram map, with missing types to fill in from 1-grams	*required*

Returns:

Type	Description
`pd.DataFrame`	new 2-gram map, with type combinations partially filled (no alias')

Source code in `nestor/keyword.py`

```
def ngram_automatch(voc1, voc2):
    """auto-match tag combinations using `nestorParams.entity_rules_map`

    Experimental method to auto-match tag combinations into higher-level
    concepts, primarily to suggest compound entity types to a user.

    Used in ``nestor.ui``

    Args:
      voc1 (pd.DataFrame): known 1-gram token->tag mapping, with types
      voc2 (pd.DataFrame): current 2-gram map, with missing types to fill in from 1-grams

    Returns:
      pd.DataFrame: new 2-gram map, with type combinations partially filled (no alias')

    """

    NE_map = nestorParams.entity_rules_map

    vocab = voc1.copy()
    vocab.NE.replace("", np.nan, inplace=True)

    # first we need to substitute alias' for their NE identifier
    NE_dict = vocab.NE.fillna("NA").to_dict()

    NE_dict.update(
        vocab.fillna("NA")
        .reset_index()[["NE", "alias"]]
        .drop_duplicates()
        .set_index("alias")
        .NE.to_dict()
    )

    _ = NE_dict.pop("", None)

    NE_text = regex_thesaurus_normalizer(NE_dict, voc2.index)

    # now we have NE-soup/DNA of the original text.
    mask = voc2.alias.replace(
        "", np.nan
    ).isna()  # don't overwrite the NE's the user has input (i.e. alias != NaN)
    voc2.loc[mask, "NE"] = NE_text[mask].tolist()

    # track all combinations of NE types (cartesian prod)

    # apply rule substitutions that are defined
    voc2.loc[mask, "NE"] = voc2.loc[mask, "NE"].apply(
        lambda x: NE_map.get(x, "")
    )  # TODO ne_sub matching issue??  # special logic for custom NE type-combinations (config.yaml)

    return voc2
```

`ngram_keyword_pipe(raw_text, vocab, vocab2)`

Experimental pipeline for one-shot n-gram extraction from raw text.

Parameters:

Name	Type	Description	Default
`raw_text`			*required*
`vocab`			*required*
`vocab2`			*required*

Source code in `nestor/keyword.py`

```
def ngram_keyword_pipe(raw_text, vocab, vocab2):
    """Experimental pipeline for one-shot n-gram extraction from raw text.

    Args:
      raw_text:
      vocab:
      vocab2:

    Returns:

    """
    import warnings

    warnings.warn(
        "This function is deprecated! Use `ngram_vocab_builder`.",
        DeprecationWarning,
        stacklevel=2,
    )
    print("calculating the extracted tags and statistics...")
    # do 1-grams
    print("\n ONE GRAMS...")
    tex = TokenExtractor()
    tex2 = TokenExtractor(ngram_range=(2, 2))
    tex.fit(raw_text)  # bag of words matrix.
    tag1_df = tag_extractor(tex, raw_text, vocab_df=vocab.loc[vocab.alias.notna()])
    vocab_combo, tex3, r1, r2 = ngram_vocab_builder(raw_text, vocab, init=vocab2)

    tex2.fit(r1)
    tag2_df = tag_extractor(tex2, r1, vocab_df=vocab2.loc[vocab2.alias.notna()])
    tag3_df = tag_extractor(
        tex3,
        r2,
        vocab_df=vocab_combo.loc[vocab_combo.index.isin(vocab2.alias.dropna().index)],
    )

    tags_df = tag1_df.combine_first(tag2_df).combine_first(tag3_df)

    relation_df = pick_tag_types(tags_df, nestorParams.derived)

    tag_df = pick_tag_types(tags_df, nestorParams.atomics + nestorParams.holes + ["NA"])
    return tag_df, relation_df
```

`ngram_vocab_builder(raw_text, vocab1, init=None)`

complete pipeline for constructing higher-order tags

A useful technique for analysts is to use their tags like lego-blocks, building up compound concepts from atomic tags. Nestor calls these *derived* entities, and are determined by `nestorParams.derived`. It is possible to construct new derived types on the fly whenever atomic or derived types are encountered together that match a "rule" set forth by the user. These are found in `nestorParams.entity_rules_map`.

Doing this in pandas and sklearn requires a bit of maneuvering with the `TokenExtractor` objects, `token_to_alias`, and `ngram_automatch`. The behavior of this function is to either produce a new ngram list from scratch using the 1-grams and the original raw-text, or to take existing n-gram mappings and add novel derived types to them.

This is a high-level function that may hide a lot of the other function calls. IT MAY SLOW DOWN YOUR CODE. The primary use is within interactive UIs that require a stream of new suggested derived-type instances, given user activity making new atomic instances.

Parameters:

Name	Type	Description	Default
`raw_text(pd.Series)`		original merged text (output from `NLPSelect`)	*required*
`vocab1(pd.DataFrame)`		known 1-gram token->tag mapping (w/ aliases)	*required*
`init(pd.DataFrame)`		2-gram mapping, known a priori (could be a prev. output of this function., optional): (Default value = None)	*required*

Returns:

Type	Description
`(tuple)`	tuple contaning: vocab2(pd.DataFrame): new/updated n-gram mapping tex(TokenExtractor): now-trained transformer that contains n-gram tf-idf scores, etc. replaced_text(pd.Series): raw text whose 1-gram tokens have been replaced with known tags replaced_again(pd.Series): replaced_text whose atomic tags have been replaced with known derived types.

Source code in `nestor/keyword.py`

```
def ngram_vocab_builder(raw_text, vocab1, init=None):
    """complete pipeline for constructing higher-order tags

    A useful technique for analysts is to use their tags like lego-blocks,
    building up compound concepts from atomic tags. Nestor calls these *derived*
    entities, and are determined by `nestorParams.derived`. It is possible to
    construct new derived types on the fly whenever atomic or derived types are
    encountered together that match a "rule" set forth by the user. These are
    found in `nestorParams.entity_rules_map`.

    Doing this in pandas and sklearn requires a bit of maneuvering with the
    `TokenExtractor` objects, `token_to_alias`, and `ngram_automatch`.
    The behavior of this function is to either produce a new ngram list from
    scratch using the 1-grams and the original raw-text, or to take existing
    n-gram mappings and add novel derived types to them.

    This is a high-level function that may hide a lot of the other function calls.
    IT MAY SLOW DOWN YOUR CODE. The primary use is within interactive UIs that
    require a stream of new suggested derived-type instances, given user
    activity making new atomic instances.

    Args:
      raw_text(pd.Series): original merged text (output from `NLPSelect`)
      vocab1(pd.DataFrame): known 1-gram token->tag mapping (w/ aliases)
      init(pd.DataFrame): 2-gram mapping, known a priori (could be a prev. output of this function., optional):  (Default value = None)

    Returns:
      (tuple): tuple contaning:
         vocab2(pd.DataFrame): new/updated n-gram mapping
         tex(TokenExtractor): now-trained transformer that contains n-gram tf-idf scores, etc.
         replaced_text(pd.Series): raw text whose 1-gram tokens have been replaced with known tags
         replaced_again(pd.Series): replaced_text whose atomic tags have been replaced with known derived types.

    """
    # raw_text, with token-->alias replacement
    replaced_text = token_to_alias(raw_text, vocab1)

    if init is None:
        tex = TokenExtractor(ngram_range=(2, 2))  # new extractor (note 2-gram)
        tex.fit(replaced_text)
        vocab2 = generate_vocabulary_df(tex)
        replaced_again = None
    else:
        mask = (np.isin(init.NE, nestorParams.atomics)) & (init.alias != "")
        # now we need the 2grams that were annotated as 1grams
        replaced_again = token_to_alias(
            replaced_text,
            pd.concat([vocab1, init[mask]])
            .reset_index()
            .drop_duplicates(subset=["tokens"])
            .set_index("tokens"),
        )
        tex = TokenExtractor(ngram_range=(2, 2))
        tex.fit(replaced_again)
        new_vocab = generate_vocabulary_df(tex, init=init)
        vocab2 = (
            pd.concat([init, new_vocab])
            .reset_index()
            .drop_duplicates(subset=["tokens"])
            .set_index("tokens")
            .sort_values("score", ascending=False)
        )
    return vocab2, tex, replaced_text, replaced_again
```

`pick_tag_types(tag_df, typelist)`

convenience function to pick out one entity type (top-lvl column)

tag_df (output from `tag_extractor`) contains multi-level columns. These can be unwieldy, especially if one needs to focus on a particular tag type, slicing by tag name. This function abstracts some of that logic away.

Gracefully finds columns that exist, ignoring ones you want that don't.

Parameters:

Name	Type	Description	Default
`tag_df(pd.DataFrame)`		binary tag occurrence matrix, as output by `tag_extractor`	*required*
`typelist(List[str])`		names of entity types you want to slice from.	*required*

Returns:

Type	Description
`(pd.DataFrameo)`	a sliced copy of `tag_df`, given `typelist`

Source code in `nestor/keyword.py`

```
def pick_tag_types(tag_df, typelist):
    """convenience function to pick out one entity type (top-lvl column)

    tag_df (output from `tag_extractor`) contains multi-level columns. These can
    be unwieldy, especially if one needs to focus on a particular tag type,
    slicing by tag name. This function abstracts some of that logic away.

    Gracefully finds columns that exist, ignoring ones you want that don't.

    Args:
      tag_df(pd.DataFrame): binary tag occurrence matrix, as output by `tag_extractor`
      typelist(List[str]): names of entity types you want to slice from.

    Returns:
      (pd.DataFrameo): a sliced copy of `tag_df`, given `typelist`

    """
    df_types = list(tag_df.columns.levels[0])
    available = set(typelist) & set(df_types)
    return tag_df.loc[:, list(available)]
```

`regex_match_vocab(vocab_iter, tokenize=False)`

regex-based multi-replace

Fast way to get all matches for a list of vocabulary (e.g. to replace them with preferred labels).

NOTE: This will avoid nested matches by sorting the vocabulary by length! This means ambiguous substring matches will default to the longest match, only.

> e.g. with vocabulary `['these','there', 'the']` and text `'there-in'` the match will defer to `there` rather than `the`.

Parameters:

Name	Type	Description	Default
`vocab_iter`	`Iterable[str]`	container of strings. If a dict is pass, will operate on keys.	*required*
`tokenize`	`bool`	whether the vocab should include all valid token strings from tokenizer	`False`

Returns:

Type	Description
`Pattern`	re.Pattern: a compiled regex pattern for finding all vocabulary.

Source code in `nestor/keyword.py`

```
def regex_match_vocab(vocab_iter, tokenize=False) -> re.Pattern:
    """regex-based multi-replace

    Fast way to get all matches for a list of vocabulary (e.g. to replace them with preferred labels).

    NOTE: This will avoid nested matches by sorting the vocabulary by length! This means ambiguous substring
    matches will default to the longest match, only.

    > e.g. with vocabulary `['these','there', 'the']` and text `'there-in'`
    > the match will defer to `there` rather than `the`.
    Args:
      vocab_iter (Iterable[str]): container of strings. If a dict is pass, will operate on keys.
      tokenize (bool): whether the vocab should include all valid token strings from tokenizer

    Returns:
      re.Pattern: a compiled regex pattern for finding all vocabulary.
    """
    sort = sorted(vocab_iter, key=len, reverse=True)
    vocab_str = r"\b(?:" + r"|".join(map(re.escape, sort)) + r")\b"

    if (not sort) and tokenize:  # just do tokenizer
        return nestorParams.token_pattern
    elif not sort:
        rx_str = r"(?!x)x"  # match nothing, ever
    elif tokenize:
        # the non-compiled token_pattern version accessed by __getitem__ (not property/attr)
        rx_str = r"({}|{})".format(
            vocab_str, r"(?:" + nestorParams["token_pattern"] + r")",
        )
    else:  # valid vocab -> match them in order of len
        rx_str = r"\b(" + "|".join(map(re.escape, sort)) + r")\b"

    return re.compile(rx_str)
```

`regex_thesaurus_normalizer(thesaurus, text)`

Quick way to replace text substrings in a Series with a dictionary of replacements (thesaurus)

Source code in `nestor/keyword.py`

```
def regex_thesaurus_normalizer(thesaurus: dict, text: pd.Series) -> pd.Series:
    """Quick way to replace text substrings in a Series with a dictionary of replacements (thesaurus)"""
    rx = regex_match_vocab(thesaurus)
    clean_text = text.str.replace(rx, lambda match: thesaurus.get(match.group(0)))
    return clean_text
```

`tag_extractor(transformer, raw_text, vocab_df=None, readable=False, group_untagged=True)`

Turn TokenExtractor instances and raw-text into binary occurrences.

Wrapper for the TokenExtractor to streamline the generation of tags from text. Determines the documents in `raw_text` that contain each of the tags in `vocab_df`, using a TokenExtractor transformer object (i.e. the tfidf vocabulary).

As implemented, this function expects an existing transformer object, though in the future this may be changed to a class-like functionality (e.g. sklearn's AdaBoostClassifier, etc) which wraps a transformer into a new one.

Parameters:

Name	Type	Description	Default
`transformer`	`object` `KeywordExtractor`	instantiated, can be pre-trained	*required*
`raw_text`	`pd.Series`	contains jargon/slang-filled raw text to be tagged	*required*
`vocab_df`	`pd.DataFrame`	An existing vocabulary dataframe or .csv filename, expected in the format of kex.generate_vocabulary_df(). (Default value = None)	`None`
`readable`	`bool`	whether to return readable, categorized, comma-sep str format (takes longer) (Default value = False)	`False`
`group_untagged`	`bool`	whether to group untagged tokens into a catch-all "_untagged" tag	`True`

Returns:

Type	Description
`pd.DataFrame`	extracted tags for each document, whether binary indicator (default) or in readable, categorized, comma-sep str format (readable=True, takes longer)

Source code in `nestor/keyword.py`

```python
def tag_extractor(
    transformer, raw_text, vocab_df=None, readable=False, group_untagged=True
):
    """Turn TokenExtractor instances and raw-text into binary occurrences.

    Wrapper for the TokenExtractor to streamline the generation of tags from text.
    Determines the documents in `raw_text` that contain each of the tags in `vocab_df`,
    using a TokenExtractor transformer object (i.e. the tfidf vocabulary).

    As implemented, this function expects an existing transformer object, though in
    the future this may be changed to a class-like functionality (e.g. sklearn's
    AdaBoostClassifier, etc) which wraps a transformer into a new one.

    Args:
       transformer (object KeywordExtractor): instantiated, can be pre-trained
       raw_text (pd.Series): contains jargon/slang-filled raw text to be tagged
       vocab_df (pd.DataFrame, optional): An existing vocabulary dataframe or .csv filename, expected in the format of
           kex.generate_vocabulary_df(). (Default value = None)
       readable (bool, optional): whether to return readable, categorized, comma-sep str format (takes longer) (Default value = False)
       group_untagged (bool, optional):  whether to group untagged tokens into a catch-all "_untagged" tag

    Returns:
       pd.DataFrame: extracted tags for each document, whether binary indicator (default)
       or in readable, categorized, comma-sep str format (readable=True, takes longer)

    """

    try:
        check_is_fitted(
            transformer._model, "vocabulary_", msg="The tfidf vector is not fitted"
        )
        toks = transformer.transform(raw_text)
    except NotFittedError:
        toks = transformer.fit_transform(raw_text)

    vocab = generate_vocabulary_df(transformer, init=vocab_df).reset_index()
    untagged_alias = "_untagged" if group_untagged else vocab["tokens"]
    v_filled = vocab.replace({"NE": {"": np.nan}, "alias": {"": np.nan}}).fillna(
        {
            "NE": "NA",  # TODO make this optional
            # 'alias': vocab['tokens'],
            # "alias": "_untagged",  # currently combines all NA into 1, for weighted sum
            "alias": untagged_alias,
        }
    )
    if group_untagged:  # makes no sense to keep NE for "_untagged" tags...
        v_filled = v_filled.assign(
            NE=v_filled.NE.mask(v_filled.alias == "_untagged", "NA")
        )
    sparse_dtype = pd.SparseDtype(int, fill_value=0.0)
    table = (  # more pandas-ey pivot, for future cat-types
        v_filled.assign(exists=1)  # placehold
        .groupby(["NE", "alias", "tokens"])["exists"]
        .sum()
        .unstack("tokens")
        .T.fillna(0)
        .astype(sparse_dtype)
    )

    A = toks[:, transformer.ranks_]
    A[A > 0] = 1

    docterm = pd.DataFrame.sparse.from_spmatrix(A, columns=v_filled["tokens"],)

    tag_df = docterm.dot(table)
    tag_df.rename_axis([None, None], axis=1, inplace=True)

    if readable:
        tag_df = _get_readable_tag_df(tag_df)

    return tag_df
```

`token_to_alias(raw_text, vocab)`

Replaces known tokens with their "tag" form

Useful if normalized text is needed, i.e. using the token->tag map from some known vocabulary list. As implemented, looks for the longest matched substrings first, ensuring precedence for compound tags or similar spellings, e.g. "thes->these" would get substituted before "the -> [article]"

Needed for higher-order tag creation (see `nestor.keyword.ngram_vocab_builder`).

Parameters:

Name	Type	Description	Default
`raw_text`	`pd.Series`	contains text with known jargon, slang, etc	*required*
`vocab`	`pd.DataFrame`	contains alias' keyed on known slang, jargon, etc.	*required*

Returns:

Type	Description
`pd.Series`	new text, with all slang/jargon replaced with unified tag representations

Source code in `nestor/keyword.py`

```
def token_to_alias(raw_text, vocab):
    """Replaces known tokens with their "tag" form

    Useful if normalized text is needed, i.e. using the token->tag map from some
    known vocabulary list. As implemented, looks for the longest matched substrings
    first, ensuring precedence for compound tags or similar spellings, e.g.
    "thes->these" would get substituted before "the -> [article]"

    Needed for higher-order tag creation (see `nestor.keyword.ngram_vocab_builder`).

    Args:
      raw_text (pd.Series): contains text with known jargon, slang, etc
      vocab (pd.DataFrame): contains alias' keyed on known slang, jargon, etc.

    Returns:
       pd.Series: new text, with all slang/jargon replaced with unified tag representations

    """
    thes_dict = vocab[vocab.alias.replace("", np.nan).notna()].alias.to_dict()
    return regex_thesaurus_normalizer(thes_dict, raw_text)
```

6.3 `nestor.datasets`

Helper function to load excavator toy dataset.

Hodkiewicz, M., and Ho, M. (2016) "Cleaning historical maintenance work order data for reliability analysis" in Journal of Quality in Maintenance Engineering, Vol 22 (2), pp. 146-163.

BscStartDate	Asset	OriginalShorttext	PMType	Cost
initialization of MWO	which excavator this MWO concerns (A, B, C, D, E)	natural language description of the MWO	repair (PM01) or replacement (PM02)	MWO expense (AUD)

Parameters:

Name	Type	Description	Default
`cleaned`	`bool`	whether to return the original dataset (False) or the dataset with keyword extraction rules applied (True), as described in Hodkiewicz and Ho (2016)	`False`

Returns:

Type	Description
`pandas.DataFrame`	raw data for use in testing nestor and subsequent workflows

Source code in `nestor/datasets/excavators.py`

```
def load_excavators(cleaned=False):
    """
    Helper function to load excavator toy dataset.

    Hodkiewicz, M., and Ho, M. (2016)
    "Cleaning historical maintenance work order data for reliability analysis"
    in Journal of Quality in Maintenance Engineering, Vol 22 (2), pp. 146-163.

    BscStartDate| Asset | OriginalShorttext | PMType | Cost
    --- | --- | --- | --- | ---
    initialization of MWO | which excavator this MWO concerns (A, B, C, D, E)| natural language description of the MWO| repair (PM01) or replacement (PM02) | MWO expense (AUD)

    Args:
        cleaned (bool): whether to return the original dataset (False) or the dataset with
            keyword extraction rules applied (True), as described in Hodkiewicz and Ho (2016)

    Returns:
        pandas.DataFrame: raw data for use in testing nestor and subsequent workflows

    """

    csv_filename = _download_excavators(cleaned=cleaned)

    df = (
        pd.read_csv(
            csv_filename, parse_dates=["BscStartDate"], sep=",", escapechar="\\"
        )
        .astype(
            {
                "Asset": AssetType,
                "OriginalShorttext": pd.StringDtype(),
                "PMType": PMType,
                "Cost": float,
            }
        )
        .rename_axis("ID")
    )

    return df
```