Published in Towards Data Science

Jessica Miles Follow
Jul 29, 2021 · 11 min read · Listen

Getting the Most out of scikit-learn Pipelines

Advanced techniques to help you combine transformation and modeling parameters in a single grid search

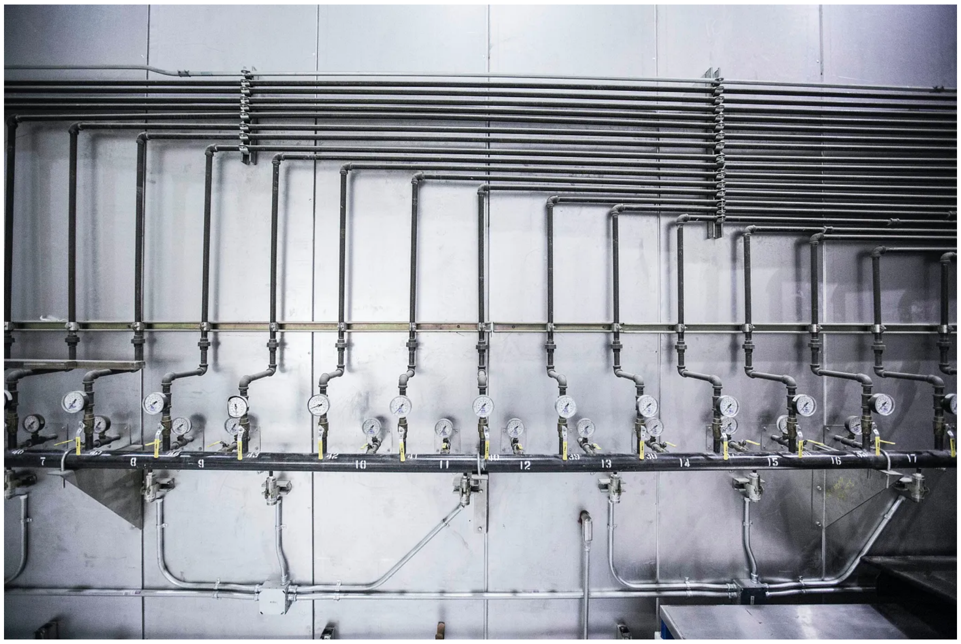


Photo by SpaceX from Pexels

`Pipelines` are extremely useful and versatile objects in the scikit-learn package. They can be nested and combined with other sklearn objects to create repeatable and easily customizable data transformation and modeling workflows.

One of the most useful things you can do with a `Pipeline` is to chain data transformation steps together with an estimator (model) at the end. You can then pass this composite estimator to a `GridSearchCV` object and search over parameters for transformation as well as model hyper-parameters in one shot. But it takes a bit of practice to learn how to construct these objects, as well as how to get and set properties in the different levels.

This post will use an NLP classification example to demonstrate how to combine a `ColumnTransformer` with a `Pipeline` and `GridSearchCV`. We'll cover some specific techniques and tips, such as how to:

- Use a `ColumnTransformer` as a `Pipeline` step to transform different types of data columns in different ways

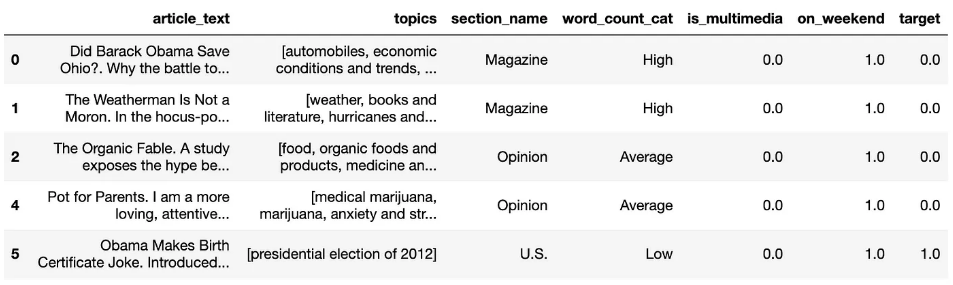
- Specify complex sets of grid search parameters to be applied together (such as binary text vectorization OR Tf-Idf normalization, but not both)
- Bypass or skip a step in the `Pipeline` using `passthrough`
- Use `set_params()` to set individual parameters in the `Pipeline` on the fly to test them outside `GridSearchCV`
- Retrieve feature names from the depths of the best estimator, for interpretation

Example Data Introduction

I'll use example data that I put together for a previous project, consisting of article text and metadata for a set of articles that The New York Times posted to Facebook.

The data has already been cleaned, but still needs to be transformed prior to modeling. This is a good example for us, because there are several different types of features that will need to be transformed in different ways:

- `article_text` : Text to be tokenized and vectorized
- `topics` : A column containing a list of applicable topics for each article, to be converted into individual features per topic
- The rest are categorical features that we will need to one-hot encode



	article_text	topics	section_name	word_count_cat	is_multimedia	on_weekend	target
0	Did Barack Obama Save Ohio?. Why the battle to...	[automobiles, economic conditions and trends, ...	Magazine	High	0.0	1.0	0.0
1	The Weatherman Is Not a Moron. In the hocus-po...	[weather, books and literature, hurricanes and...	Magazine	High	0.0	1.0	0.0
2	The Organic Fable. A study exposes the hype be...	[food, organic foods and products, medicine an...	Opinion	Average	0.0	1.0	0.0
4	Pot for Parents. I am a more loving, attentive...	[medical marijuana, marijuana, anxiety and str...	Opinion	Average	0.0	1.0	0.0
5	Obama Makes Birth Certificate Joke. Introduced...	[presidential election of 2012]	U.S.	Low	0.0	1.0	1.0

The columns in our example data set, stored in a dataframe

The `target` is already binarized here, with 0 indicating reader had low engagement with the Facebook post, and 1 indicating high engagement. The goal of this example classification problem will be to predict engagement level.

To prepare, we'll assign the different sets of column names to variables for flexibility, and separate our 93 7 t sets.

```python
from sklearn.model_selection import train_test_split

# sets of columns to be transformed different ways
text_col = 'article_text'
topic_col = 'topics'
cat_cols = ['section_name', 'word_count_cat', 'is_multimedia', 'on_weekend']

# initial X and y set for all posts
X = df[[text_col, topic_col] + cat_cols]
y = df['target']

# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2, stratify=y)
```

pipelines_tts.py hosted with ❤ by GitHub view raw

Perform train-test-split and create variables for different sets of columns

Build ColumnTransformer for Transformation

First, we're going to create a `ColumnTransformer` to transform the data for modeling. We'll use `ColumnTransformer` for this instead of a `Pipeline` because it allows us to specify different transformation steps for different columns, but results in a single matrix of features.

As a reminder (or introduction, if you haven't used them before) regular `Pipelines` take a list of tuples as input, where the first value in each tuple is the step name, and the second value is the estimator object.

```python
pipe = Pipeline([
    ('vect', CountVectorizer()),
    ('clf', LogisticRegression())
])
```

`ColumnTransformers` are built similarly to `Pipelines`, except **you include a third value in each tuple representing the columns to be transformed in that step**. Since our data is in a DataFrame, we can pass strings and lists of strings representing the DataFrame column names. If your data is stored in an array, you can pass a column index or array of column indices.

I've already stored the names of the different categories of columns as variables, so they can be dynamically passed to the `ColumnTransformer` like so:

```
cols_trans = ColumnTransformer([
    ('txt', TfidfVectorizer(), text_col),
    ('txt_kw', CountVectorizer(), topic_col),
    ('ohe', OneHotEncoder(drop='first'), cat_cols),
    remainder='drop'
])
```

`Pipeline` steps are executed serially, where the output from the first step is passed to the second step, and so on. **`ColumnTransformers` are different in that each step is executed separately, and the transformed features are concatenated at the end.** This saves us from having to do the concatenation ourselves, and will also make it easy to get the full list of feature names when we're ready to interpret our best model.

By default, any columns you pass into the `ColumnTransformer` that aren't specified to be transformed will be dropped (`remainder='drop'`). **If you have columns that you want to include but do not need to be transformed, specify** `remainder='passthrough'`. We'll see the `passthrough` parameter again later, as it can used in other contexts in sklearn to skip or bypass a processing step.

Here's the full code for this step in our example workflow, with explanation below:

```python
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# Create a custom pass-through function for the topics, since they are
# already tokenized. We will only use CountVectorizer to split the topics
# into separate feature columns.
def no_analyzer(doc):
    """Pass-through function to avoid transforming topics lists, which are
    already tokenized.
    """
    return doc

# create ColumnTransformer, and pass the column names to transform in each step
cols_trans = ColumnTransformer([
    ('txt', TfidfVectorizer(), text_col),
    ('txt_kw', CountVectorizer(analyzer=no_analyzer), topic_col),
    ('ohe', OneHotEncoder(drop='first'), cat_cols)
])
```

pipelines_coltrans.py hosted with ❤ by GitHub view raw

Create ColumnTransformer to transform the different types of data columns we have

- `text_col` will be transformed using a `TfidfVectorizer`, which will tokenize the text of each document and create vectors to form a document-term matrix. We will be able to specify many different options for this transformation that will affect final model performance, such as stop words to remove, size of n-grams to generate, maximum number of features to include, and how to normalize the token counts in the document-term matrix. However, since we're going to specify different options for those parameters to be tried in our grid search, we'll just create a vanilla instance of the transformer for now.
- `topic_col` is already a list of topics in a single column, from which we want to create a binary document-term matrix. We'll use `CountVectorizer`, and give it a custom `analyzer` that doesn't do anything. The default `analyzer` usually performs preprocessing, tokenizing, and n-grams generation and outputs a list of tokens, but since we already have a list of tokens, we'll just pass them through as-is, and `CountVectorizer` will return a document-term matrix of the existing topics without tokenizing them further.
- `cat_cols` consists of our categorical columns, which we will one-hot encode using `OneHotEncoder`. The only parameter we'll specify is to drop the first category in each column, since we'll be using a regression model.

> ***Tip:*** *Both* `TfidfVectorizer` *and* `CountVectorizer` *expect a* ***1-D array****, so the column name needs to be passed to* `ColumnTransformer` *as a string and not as a list, even if the list has only a single entry. If you give either of these transformers a list, you will get an error referring to* `incompatible row dimensions`*. Most other sklearn transformers expect a 2-D array (such as* `OneHotEncoder`*), so even if you're only transforming a single column, you need to pass a list.*

Great, so now we have `cols_trans`, a `ColumnTransformer` object that will output a single feature matrix with our transformed data.

Create Pipeline with ColumnTransformer and Model

Next, we'll create a `Pipeline` where `cols_trans` is the first step, and a Logistic Regression classifier is the second step.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ('trans', cols_trans),
    ('clf', LogisticRegression(max_iter=300,
class_weight='balanced'))
])
```

If we called `pipe.fit(X_train, y_train)` , we would be transforming our `X_train` data and fitting the Logistic Regression model to it in a single step. Note that we can simply use `fit()` , and don't need to do anything special to specify we want to both fit AND transform the data in the first step; the pipeline will know what to do.

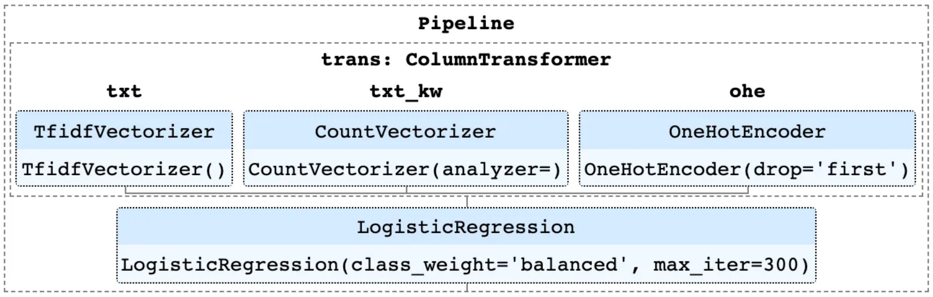
Once you start nesting `Pipelines` and other objects, you want to refresh yourself on how the steps will be executed. One way to do this is to set sklearn's `display` parameter to `'diagram'` to show an HTML representation when you call `display()` on the pipeline object itself. The HTML will be interactive in a Jupyter Notebook, and you can click on each step to expand it and see its current parameters.

```
from sklearn import set_config

set_config(display='diagram')

# with display='diagram', simply use display() to see the diagram
display(pipe)

# if desired, set display back to the default
set_config(display='text')
```



Example HTML display of Pipeline, with parameters shown

Prepare Grid Search Parameters

We will be able to pass our `pipe` object to a `GridSearchCV` to search parameters for both the transformation and the classifier model at the same time. `GridSearchCV` will want a dictionary of search parameters to try, where the keys are the pipeline steps/parameter names, and the values are lists of the parameters to be searched over.

With a `ColumnTransformer` nested in a `Pipeline` like we have, it can be tricky to get the keys of this dictionary just right, since they're named after the label of each step, with dunders `__` as separators. The easiest way to get a list of all available options is to use `pipe.get_params()`.

You should see something like this:

```
{'memory': None,
 'steps': [('trans',
   ColumnTransformer(transformers=[('txt', TfidfVectorizer(),
'article_text'),
                                   ('txt_kw',
                                    CountVectorizer(analyzer=
<function no_analyzer at 0x7fbc5f4bac10>),
                                    'topics'),
                                   ('ohe',
OneHotEncoder(drop='first'),
                                    ['section_name',
'word_count_cat',
                                     'is_multimedia',
'on_weekend'])])),
  ('clf', LogisticRegression(class_weight='balanced',
max_iter=300))],
 'verbose': False,
 'trans': ColumnTransformer(transformers=[('txt', TfidfVectorizer(),
'article_text'),
                                 ('txt_kw',
                                  CountVectorizer(analyzer=<function
no_analyzer at 0x7fbc5f4bac10>),
                                  'topics'),
                                 ('ohe', OneHotEncoder(drop='first'),
                                  ['section_name', 'word_count_cat',
                                   'is_multimedia', 'on_weekend'])]),
 'clf': LogisticRegression(class_weight='balanced', max_iter=300),
 'trans__n_jobs': None,
 'trans__remainder': 'drop',
 'trans__sparse_threshold': 0.3,
 'trans__transformer_weights': None,
 'trans__transformers': [('txt', TfidfVectorizer(), 'article_text'),
  ('txt_kw',
   CountVectorizer(analyzer=<function no_analyzer at
0x7fbc5f4bac10>),
   'topics'),
  ('ohe',
   OneHotEncoder(drop='first'),
   ['section_name', 'word_count_cat', 'is_multimedia',
'on_weekend'])],
 'trans__verbose': False,
 'trans__txt': TfidfVectorizer(),
 'trans__txt_kw': CountVectorizer(analyzer=<function no_analyzer at
0x7fbc5f4bac10>),
 'trans__ohe': OneHotEncoder(drop='first'),
 'trans__txt__analyzer': 'word',
 'trans__txt__binary': False,
 'trans__txt__decode_error': 'strict',
 'trans__txt__dtype': numpy.float64,
 'trans__txt__encoding': 'utf-8',
 'trans__txt__input': 'content',
 'trans__txt__lowercase': True,
 'trans__txt__max_df': 1.0,
 'trans__txt__max_features': None,
 'trans__txt__min_df': 1,
 'trans__txt__ngram_range': (1, 1),
 'trans__txt__norm': 'l2',
 'trans__txt__preprocessor': None,
 'trans__txt__smooth_idf': True,
 'trans__txt__stop_words': None,
 'trans__txt__strip_accents': None,
 'trans__txt__sublinear_tf': False,
 'trans__txt__token_pattern': '(?u)\\b\\w\\w+\\b',
 'trans__txt__tokenizer': None,
 'trans__txt__use_idf': True,
 'trans__txt__vocabulary': None,
 'trans__txt_kw__analyzer': <function __main__.no_analyzer(doc)>,
 'trans__txt_kw__binary': False,
 'trans__txt_kw__decode_error': 'strict',
 'trans__txt_kw__dtype': numpy.int64,
 'trans__txt_kw__encoding': 'utf-8',
 'trans__txt_kw__input': 'content',
 'trans__txt_kw__lowercase': True,
 'trans__txt_kw__max_df': 1.0,
 'trans__txt_kw__max_features': None,
 'trans__txt_kw__min_df': 1,
 'trans__txt_kw__ngram_range': (1, 1),
 'trans__txt_kw__preprocessor': None,
 'trans__txt_kw__stop_words': None,
 'trans__txt_kw__strip_accents': None,
 'trans__txt_kw__token_pattern': '(?u)\\b\\w\\w+\\b',
 'trans__txt_kw__tokenizer': None,
 'trans__txt_kw__vocabulary': None,
 'trans__ohe__categories': 'auto',
 'trans__ohe__drop': 'first',
 'trans__ohe__dtype': numpy.float64,
 'trans__ohe__handle_unknown': 'error',
 'trans__ohe__sparse': True,
 'clf__C': 1.0,
 'clf__class_weight': 'balanced',
 'clf__dual': False,
 'clf__fit_intercept': True,
 'clf__intercept_scaling': 1,
 'clf__l1_ratio': None,
 'clf__max_iter': 300,
 'clf__multi_class': 'auto',
 'clf__n_jobs': None,
 'clf__penalty': 'l2',
 'clf__random_state': None,
 'clf__solver': 'lbfgs',
 'clf__tol': 0.0001,
 'clf__verbose': 0,
 'clf__warm_start': False}
```

Scroll down to the bottom of the output, and you'll see the parameters for each step listed in the exact format you'll need to pass to `GridSearchCV`. Only the current value for each parameter will be listed, so you may need to review the documentation for each estimator to see what other values are supported.

Let's say I decide want to search through the following parameters for text vectorization and my Logistic Regression model:

```
grid_params = {
    'trans__txt__binary': [True, False],
    'trans__txt__use_idf': [True, False],
    'trans__txt__max_features': [None, 100000, 10000],
    'trans__txt__ngram_range': [(1, 1), (1, 2), (1, 3)],
    'trans__txt__stop_words': [None, nltk_stopwords],
    'clf__C': [1.0, 0.1, 0.01],
    'clf__fit_intercept': [True, False],
    'clf__penalty': ['l2', 'l1'],
    'clf__solver': ['lbfgs','saga']
}
```

But what if I want to use `'trans__txt__binary': True` only with `'trans__txt__use_idf': False` , so that I really get a binary 0 or 1 output? And I want to try a regular Term Frequency by itself, as well as TF + IDF, but in that case I only want binary to be False?

If I run the search with the parameters as written above, `GridSearchCV` will try every combination, even ones that might not make sense.

It turns out, **GridSearchCV will also accept a list of dictionaries of parameters, and will be smart enough to try only the unique combinations across all the dicts in the list.** We need to put the common parameters in both, params for binary in one dictionary, and the params for regular token count in the other.

Creating search parameters that account for specific combinations I want to test

Note that this may not be necessary if you only have a few parameters you're searching, but it really becomes important when you want to try many, like we do here. Each combination will take a certain amount of computational power and time, so we don't want to run any unnecessary combinations.

How to Bypass or Skip Entire Steps

Many of the actual parameters will have a `None` value you can include if you want to test results without using that option. However, this won't work if you have a workflow where you want to **bypass or skip an entire step in the Pipeline.**

An example would be if you have continuous data and want to evaluate Linear Regression model performance using both a `MinMaxScaler` and a `StandardScaler` to see which works better. You could add each as a separate step in your `Pipeline`, and use a version of the technique above to create a list of gris parameters to try either `MinMax` or `Standard` Scaler, but not both at once. You can give `'passthrough'` as the parameter value to the named pipeline step to bypass it, so that the other scaler is the only one being used.

Use 'passthrough' to skip either MinMax or Standard Scaler, so we can test them individually in a single grid search

Note that if you simply want to try applying and not applying a given step (instead of substituting one step for another, as we have above) you can include an instance of the transformer object itself in the parameter list along with `'passthrough'`. For example, in this example from the sklearn documentation, if you have a dimensionality reduction step labeled `'pca'` for Principal Component Analysis, you could include `'pca': ['passthrough', PCA(5), PCA(10)]` in your grid search params to test using no PCA, using PCA with 5 components, and using PCA with 10 components.

Setting Pipeline Parameters Individually

OK, so assuming our grid search completed, the `gs` object will automatically have been fitted with the best estimator, which we can access using `gs.best_estimator_` (don't forget that underscore at the end; it's necessary!).

Since we gave `gs` our `pipe` object as the estimator, the fitted best estimator is a copy of `pipe`, with the best parameters applied, and already fitted on the current X_train. We can do things like:

- Call `gs.best_estimator_.get_params()` to get the parameters of the best-performing pipeline
- Export `gs.best_estimator_` to file using pickle or joblib to back up the best pipeline complete with its parameters and current fit on X_train
- Call `gs.best_estimator_.predict(X_test)` to get predictions on unseen test data
- Use `set_params()` in case we want to test tweaking any individual parameters individually

For instance, here we tested `max_features` of `[10000, 5000, 2000]` in our grid search, and `2000` performed the best. Since it's the lowest, perhaps I want to evaluate performance with only `1000` to see if going even lower would actually be better.

I could manually set a parameter value to something else using `set_params()` and a similar syntax to what we used when specifying the grid search params:

```
gs.best_estimator_.set_params(**{'trans__txt__max_features': 1000})
```

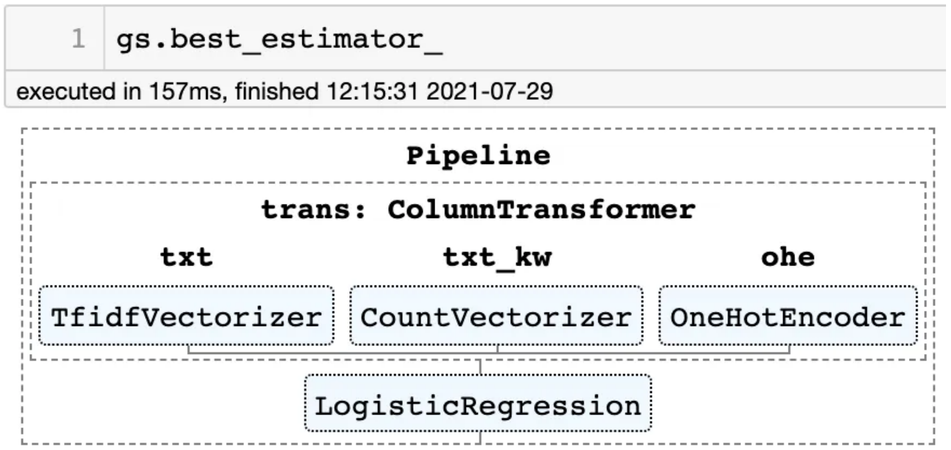
Note that I pass the parameter as a dictionary, and could include multiple parameters if needed. I also need the `**` before the dictionary to unpack it so `set_params()` will accept it.

Accessing Feature Names from ColumnTransformer

Let's assume that `max_features` of `1000` didn't help, so I've set it back to `2000`, evaluated performance on unseen test data, and am now ready to do some interpretation of the model coefficients.

To do that, I'll need to get the final transformed feature names after we tokenized the article text, split up the topics, and one-hot encoded the categorical features. We can get that pretty easily, but we'll have use the correct syntax to reach down through the nested levels.

First, let's use the HTML display trick to remind ourselves what the steps are and what they're labeled:



```
gs.best_estimator_
```

```
executed in 157ms, finished 12:15:31 2021-07-29
```

HTML display of the best estimator from GridSearch

Remember that although we named the `ColumnTransformer` object `cols_trans`, we labeled it just `trans` in the final pipeline we fed to `gs`. **So `trans` is the first label we'll need to deal with here.**

Although the transformation steps are shown in parallel here, they're actually done in an order in terms of how they're concatenated. **If we want our full list of feature names list to be accurate, we'll need to get the feature names from each transformer in the order they were originally applied, then concatenate them**. In the diagram, the correct order of that layer can be read from left to right.

We'll use the `named_steps()` property to access the transformation layer, and then get the `get_feature_names()` property from each of the `named_transformers_()` in that layer, in order.

Getting feature names from each transformer, in order, for model interpretation

Note that for the `ohe` transformer, I passed `cat_cols` to `get_feature_names()` which is the list of original column names that were transformed in this step. Passing the original column names here using them as the column prefixes; otherwise they will be generic.

Conclusion

Hopefully this has been a useful example of how to construct a nested `Pipeline` to handle both transformation and modeling. Although it takes a little extra work to build, doing so can be very beneficial if you want to try out different transformation and model parameters in your grid search.

Feedback and questions welcome!

Sklearn Pipeline NLP Data Transformation Gridsearchcv Columntransformer