How To Automate SEO Keyword Clustering By Search Intent With Python via @sejournal, @andreasvoniatis

There’s a lot to know about search intent, from using deep learning to infer search intent by classifying text and breaking down SERP titles using Natural Language Processing (NLP) techniques, to clustering based on semantic relevance, with the benefits explained.

Not only do we know the benefits of deciphering search intent, but we also have a number of techniques at our disposal for scale and automation.

So, why do we need another article on automating search intent?

Search intent is ever more important now that AI search has arrived.

While more was generally in the 10 blue links search era, the opposite is true with AI search technology, as these platforms generally seek to minimize the computing costs (per FLOP) in order to deliver the service.

SERPs Still Contain The Best Insights For Search Intent

The techniques so far involve doing your own AI, that is, getting all of the copy from titles of the ranking content for a given keyword and then feeding it into a neural network model (which you have to then build and test) or using NLP to cluster keywords.

What if you don’t have time or the knowledge to build your own AI or invoke the Open AI API?

While cosine similarity has been touted as the answer to helping SEO professionals navigate the demarcation of topics for taxonomy and site structures, I still maintain that search clustering by SERP results is a far superior method.

That’s because AI is very keen to ground its results on SERPs and for good reason – it’s modelled on user behaviors.

There is another way that uses Google’s very own AI to do the work for you, without having to scrape all the SERPs content and build an AI model.

Let’s assume that Google ranks site URLs by the likelihood of the content satisfying the user query in descending order. It follows that if the intent for two keywords is the same, then the SERPs are likely to be similar.

For years, many SEO professionals compared SERP results for keywords to infer shared (or shared) search intent to stay on top of core updates, so this is nothing new.

The value-add here is the automation and scaling of this comparison, offering both speed and greater precision.

How To Cluster Keywords By Search Intent At Scale Using Python (With Code)

Assuming you have your SERPs results in a CSV download, let’s import it into your Python notebook.

1. Import The List Into Your Python

Notebook

```
import pandas as pd
import numpy as np serps_input = pd.read_csv('data/sej_serps_input.csv')
del serps_input['Unnamed: 0']
serps_input
```

Below is the SERPs file now imported into a Pandas dataframe.

Image from author, April 2025

2. Filter Data For Page 1

We want to compare the Page 1 results of each SERP between keywords.

We’ll split the dataframe into mini keyword dataframes to run the filtering function before recombining into a single dataframe, because we want to filter at the keyword level:

```
# Split serps_grpby_keyword = serps_input.groupby("keyword")
k_urls = 15 # Apply Combine
def filter_k_urls(group_df): filtered_df = group_df.loc[group_df['url'].notnull()] filtered_df = filtered_df.loc[filtered_df['rank'] <= k_urls] return filtered_df
filtered_serps = serps_grpby_keyword.apply(filter_k_urls) # Combine
## Add prefix to column names
#normed = normed.add_prefix('normed_') # Concatenate with initial data frame
filtered_serps_df = pd.concat([filtered_serps],axis=0)
del filtered_serps_df['keyword']
filtered_serps_df = filtered_serps_df.reset_index()
del filtered_serps_df['level_1']
filtered_serps_df
```

Image from author, April 2025

3. Convert Ranking URLs To A String

Because there are more SERP result URLs than keywords, we need to compress those URLs into a single line to represent the keyword’s SERP.

Here’s how:

```
# convert results to strings using Split Apply Combine filtserps_grpby_keyword = filtered_serps_df.groupby("keyword") def string_serps(df): df['serp_string'] = ''.join(df['url']) return df # Combine strung_serps = filtserps_grpby_keyword.apply(string_serps) # Concatenate with
```

```
initial data frame and clean strung_serps = pd.concat([strung_serps],axis=0) strung_serps = strung_serps[['keyword', 'serp_string']]#.head(30) strung_serps = strung_serps.drop_duplicates() strung_serps
```

Below shows the SERP compressed into a single line for each keyword.

Image from author, April 2025

4. Compare SERP Distance

To perform the comparison, we now need every combination of keyword SERP paired with other pairs:

```
# align serps
```

```
def serps_align(k, df): prime_df = df.loc[df.keyword == k] prime_df = prime_df.rename(columns = {"serp_string" : "serp_string_a", 'keyword': 'keyword_a'}) comp_df = df.loc[df.keyword != k].reset_index(drop=True) prime_df = prime_df.loc[prime_df.index.repeat(len(comp_df.index))].reset_index(drop=True) prime_df = pd.concat([prime_df, comp_df], axis=1) prime_df = prime_df.rename(columns = {"serp_string" : "serp_string_b", 'keyword': 'keyword_b', "serp_string_a" : "serp_string", 'keyword_a': 'keyword'}) return prime_df columns = ['keyword', 'serp_string', 'keyword_b', 'serp_string_b']
matched_serps = pd.DataFrame(columns=columns)
matched_serps = matched_serps.fillna(0)
queries = strung_serps.keyword.to_list() for q in queries: temp_df = serps_align(q, strung_serps) matched_serps = matched_serps.append(temp_df) matched_serps
```

The above shows all of the keyword SERP pair combinations, making it ready for SERP string comparison.

There is no open-source library that compares list objects by order, so the function has been written for you below.

The function “serp_compare” compares the overlap of sites and the order of those sites between SERPs.

```
import py_stringmatching as sm
ws_tok = sm.WhitespaceTokenizer() # Only compare the top k_urls results def serps_similarity(serps_str1, serps_str2, k=15): denom = k+1 norm = sum([2*(1/i - 1.0/(denom)) for i in range(1, denom)]) #use to tokenize the URLs ws_tok =
```

```
sm.WhitespaceTokenizer() #keep only first k URLs serps_1 = ws_tok.tokenize(serps_str1)[:k] serps_2 = ws_tok.tokenize(serps_str2)[:k] #get positions of matches match = lambda a, b: [b.index(x)+1 if x in b else None for x in a] #positions intersections of form [(pos_1, pos_2), ...] pos_intersections = [(i+1,j) for i,j in enumerate(match(serps_1, serps_2)) if j is not None] pos_in1_not_in2 = [i+1 for i,j in enumerate(match(serps_1, serps_2)) if j is None] pos_in2_not_in1 = [i+1 for i,j in enumerate(match(serps_2, serps_1)) if j is None] a_sum = sum([abs(1/i -1/j) for i,j in pos_intersections]) b_sum = sum([abs(1/i -1/denom) for i in pos_in1_not_in2]) c_sum = sum([abs(1/i -1/denom) for i in pos_in2_not_in1]) intent_prime = a_sum + b_sum + c_sum intent_dist = 1 - (intent_prime/norm) return intent_dist # Apply the function matched_serps['si_simi'] = matched_serps.apply(lambda x: serps_similarity(x.serp_string, x.serp_string_b), axis=1) # This is what you get matched_serps[['keyword', 'keyword_b', 'si_simi']]
```

Now that the comparisons have been executed, we can start clustering keywords.

We will be treating any keywords that have a weighted similarity of 40% or more.

```
# group keywords by search intent
simi_lim = 0.4 # join search volume
keysv_df           =           serps_input[['keyword',
'search_volume']].drop_duplicates()
keysv_df.head() # append topic vols
keywords_crossed_vols = serps_compared.merge(keysv_df, on =
'keyword', how = 'left')
keywords_crossed_vols = keywords_crossed_vols.rename(columns =
```

```
{'keyword': 'topic', 'keyword_b': 'keyword', 'search_volume': 'topic_volume'}) # sim si_simi
keywords_crossed_vols.sort_values('topic_volume', ascending = False) # strip NAN
keywords_filtered_nonnan = keywords_crossed_vols.dropna()
keywords_filtered_nonnan
```

We now have the potential topic name, keywords SERP similarity, and search volumes of each.

You'll note that keyword and keyword_b have been renamed to topic and keyword, respectively.

Now we're going to iterate over the columns in the dataframe using the lambda technique.

The lambda technique is an efficient way to iterate over rows in a Pandas dataframe because it converts rows to a list as opposed to the .iterrows() function.

Here goes:

```
queries_in_df = list(set(matched_serps['keyword'].to_list()))
topic_groups = {} def dict_key(dicto, keyo): return keyo in
dicto def dict_values(dicto, vala): return any(vala in val for
val in dicto.values()) def what_key(dicto, vala): for k, v in
dicto.items(): if vala in v: return k def find_topics(si,
keyw, topc): if (si >= simi_lim): if (not
dict_key(sim_topic_groups, keyw)) and (not
dict_key(sim_topic_groups, topc)): if (not
dict_values(sim_topic_groups, keyw)) and (not
dict_values(sim_topic_groups, topc)): sim_topic_groups[keyw] =
[keyw] sim_topic_groups[keyw] = [topc] if
dict_key(non_sim_topic_groups, keyw):
non_sim_topic_groups.pop(keyw) if
dict_key(non_sim_topic_groups, topc):
non_sim_topic_groups.pop(topc) if
(dict_values(sim_topic_groups, keyw)) and (not
dict_values(sim_topic_groups, topc)): d_key =
what_key(sim_topic_groups, keyw)
sim_topic_groups[d_key].append(topc) if
dict_key(non_sim_topic_groups, keyw):
non_sim_topic_groups.pop(keyw) if
dict_key(non_sim_topic_groups, topc):
non_sim_topic_groups.pop(topc) if (not
dict_values(sim_topic_groups, keyw)) and
(dict_values(sim_topic_groups, topc)): d_key =
what_key(sim_topic_groups, topc)
sim_topic_groups[d_key].append(keyw) if
dict_key(non_sim_topic_groups, keyw):
non_sim_topic_groups.pop(keyw) if
dict_key(non_sim_topic_groups, topc):
non_sim_topic_groups.pop(topc) elif (keyw in sim_topic_groups)
and (not topc in sim_topic_groups):
sim_topic_groups[keyw].append(topc)
sim_topic_groups[keyw].append(keyw) if keyw in
```

```
non_sim_topic_groups: non_sim_topic_groups.pop(keyw) if topc in non_sim_topic_groups: non_sim_topic_groups.pop(topc) elif (not keyw in sim_topic_groups) and (topc in sim_topic_groups): sim_topic_groups[topc].append(keyw) sim_topic_groups[topc].append(topc) if keyw in non_sim_topic_groups: non_sim_topic_groups.pop(keyw) if topc in non_sim_topic_groups: non_sim_topic_groups.pop(topc) elif (keyw in sim_topic_groups) and (topc in sim_topic_groups): if len(sim_topic_groups[keyw]) > len(sim_topic_groups[topc]): sim_topic_groups[keyw].append(topc) [sim_topic_groups[keyw].append(x) for x in sim_topic_groups.get(topc)] sim_topic_groups.pop(topc) elif len(sim_topic_groups[keyw]) < len(sim_topic_groups[topc]): sim_topic_groups[topc].append(keyw) [sim_topic_groups[topc].append(x) for x in sim_topic_groups.get(keyw)] sim_topic_groups.pop(keyw) elif len(sim_topic_groups[keyw]) == len(sim_topic_groups[topc]): if sim_topic_groups[keyw] == topc and sim_topic_groups[topc] == keyw: sim_topic_groups.pop(keyw) elif si < simi_lim: if (not dict_key(non_sim_topic_groups, keyw)) and (not dict_key(sim_topic_groups, keyw)) and (not dict_values(sim_topic_groups,keyw)): non_sim_topic_groups[keyw] = [keyw] if (not dict_key(non_sim_topic_groups, topc)) and (not dict_key(sim_topic_groups, topc)) and (not dict_values(sim_topic_groups,topc)): non_sim_topic_groups[topc] = [topc]
```

Below shows a dictionary containing all the keywords clustered by search intent into numbered groups:

```
{1: ['fixed rate isa', 'isa rates', 'isa interest rates', 'best isa rates', 'cash isa', 'cash isa rates'], 2: ['child savings account', 'kids savings account'], 3: ['savings account', 'savings account interest rate', 'savings rates', 'fixed rate savings', 'easy access savings', 'fixed rate bonds', 'online savings account', 'easy access savings account', 'savings accounts uk'], 4: ['isa account', 'isa', 'isa savings']}
```

Let’s stick that into a dataframe:

```
topic_groups_lst = [] for k, l in topic_groups_numbered.items(): for v in l: topic_groups_lst.append([k, v]) topic_groups_dictdf = pd.DataFrame(topic_groups_lst, columns=['topic_group_no', 'keyword']) topic_groups_dictdf
```

Image from author, April 2025

The search intent groups above show a good approximation of the keywords inside them, something that an SEO expert would likely achieve.

Although we only used a small set of keywords, the method can obviously be scaled to thousands (if not more).

Activating The Outputs To Make Your Search Better

Of course, the above could be taken further using neural networks, processing the ranking content for more accurate clusters and cluster group naming, as some of the commercial products out there already do.

For now, with this output, you can:

- Incorporate this into your own SEO dashboard systems to make your trends and SEO reporting more meaningful.
- Build better paid search campaigns by structuring your Google Ads accounts by search intent for a higher Quality Score.
- Merge redundant facet ecommerce search URLs.
- Structure a shopping site's taxonomy according to search intent instead of a typical product catalog.

I'm sure there are more applications that I haven't mentioned – feel free to comment on any important ones that I've not already mentioned.

In any case, your SEO keyword research just got that little bit more scalable, accurate, and quicker!

Download the full code here for your own use.

More Resources:

Featured Image: Buch and Bee/Shutterstock

https://www.searchenginejournal.com/how-to-automate-seo-keyword-clustering-by-search-intent/539347/