**Building a Basic Machine Learning Model in Python**

***Extensive essay on how to pick the right problem and how to develop a basic classifier***

By now, all of us have seen the results of various basic machine learning (ML) models. The internet is rife with images, videos, and articles showing off how a computer identifies, correctly or not, various animals.

While we have moved towards more intricate machine learning models, such as ones that generate or upscale images, those basic ones still form the foundation of those efforts. Mastering the basics can become a launchpad for much greater future endeavors.

So, I decided to revisit the basics myself and build a basic machine learning model with several caveats — it must be somewhat useful, as simplistic as possible, and return reasonably accurate results.

Unlike many other tutorials on the internet, however, I want to present my entire thought process from beginning to end. As such, the coding part will begin quite a bit later as problem selection in both the theoretical and practical realm is equally important. In the end, I believe that understanding *why* will go further than *how to*.

**Picking the correct problem for ML**

Although machine learning can solve a great deal of challenges, it’s not a one-size-fits-all approach. Even if we were to temporarily forget about the financial, temporal, and other resource costs, ML models would still be great at some things and terrible at others.

Categorization is a great example of where machine learning may shine. Whenever we deal with real world data (i.e., we’re not dealing with categories created within the code itself), figuring out all possible rules that define a phenomenon is nearly impossible.

As I’ve written previously, if we were to attempt to take a rule-based approach to categorize whether an object is a cat or not, we’d quickly run into issues. There seems to be no defining quality that makes any physical object what it is — there are cats without tails, fur, ears, one eye, a different number of legs, etc., but all of them still fall within the same category.

Enumerating all of the possible rules and exceptions to them is likely impossible, maybe there even isn’t some eternal list, and we make them up as we go. Machine learning, in some sense, mimics our thinking by eating up an enormous amount of data to make predictions.

In other words, we should carefully consider the problem we’re trying to solve before trying to figure out which model would fit best, how much data we’ll need, and many other things we concern ourselves with once we start the task.

**In search of practical application**

Making models that differentiate between dogs and cats is certainly interesting and fun but unlikely to net any benefit, even if we scale up the operation to immense levels. Additionally, there have been millions of tutorials for such models created online.

I decided to pick word categorization, as it hasn’t been as frequently written about, and it has some practical application. Our SEO team had an interesting proposition — they needed to categorize keywords according to three types:

1. **Informational** — users searching for knowledge about a topic (e.g., “what is a proxy”)
2. **Transactional** — users seeking for a product or service (e.g., “best proxies”)
3. **Navigational** — users seeking for a specific brand or an offshoot of it (e.g., “Oxylabs)

Categorizing thousands of keywords manually is a bit of a pain. Such a task seems (almost) perfect for machine learning, although there’s an inherent issue that is nearly impossible to solve, which I will expand upon later.

Finally, it made data collection and management a significantly easier task than it would otherwise have been. SEO specialists use a variety of tools to track keywords, most of which can export thousands of them into a CSV sheet. All that needs to be done is to assign categories to the keywords.

**Building a pre-MVP**

Deciding how many data points you’ll need before building a model is nearly impossible. There are some dependencies on the stated goal (i.e., more or less categories), however, calculating these with precision is a fool’s errand. Picking a sufficiently large number (e.g., 1000 entries) is a good starting point.

One thing I’d caution against is working with the entire dataset first. Since it is likely, it’s the first time you’re developing a model, a lot of things can go wrong. In general, you’re better off writing the code and running it on a small sample (e.g., 10% of the total) just to ensure there are no semantic errors or any other horrors.

Once you get the desired result, start working with the entire dataset. While it’s unlikely that you’ll have to throw out the project entirely, you don’t want to end up spending hours of (boring) work and have nothing to show for.

Regardless, with some samples in hand, we can begin the development experience properly. I’ve chosen Python as it’s a fairly common language with decent support for machine learning through its numerous libraries.

**Libraries**

1. [Pandas](https://pypi.org/project/pandas/). While not strictly necessary, reading and exporting to CSV is going to make our lives significantly easier.
2. [SciKit-Learn](https://pypi.org/project/scikit-learn/). A fairly powerful and flexible machine learning library, which will form the foundation for our classification model. We’ll be using various *sklearn* features throughout the tutorial.
3. [NLTK](https://pypi.org/project/nltk/) (Natural Language Toolkit). As we’ll be processing natural language, NLTK does the job perfectly. *Stopwords* will be absolutely necessary from the package.

**Imports**

import pandas as pd  
from sklearn.feature\_extraction.text import TfidfVectorizer  
from sklearn.linear\_model import LogisticRegression  
from sklearn.pipeline import Pipeline  
from sklearn.feature\_selection import SelectKBest, chi2  
from nltk.corpus import stopwords

**Line 1**

Fairly self-explanatory. *Pandas* allows us to read and write CSV and other spreadsheet files by creating data frames. Since we’ll be dealing with keywords, most SEO tools export lists of them in CSV, which will reduce the data processing we need to do manually.

**Line 2**

From the SciKit-Learn library, we’ll pick up several things, *TfidfVectorizer* being our first choice.

Vectorizers convert our strings into feature vectors, which results in two important changes. First, strings are converted into numerical representations. Each unique string is converted into an index, which is then turned into a vector (the offshoot of a matrix).

**Sentence #1**: “The dog is brown.”

**Sentence #2**: “The dog is black.”

Vectorization would take both sentences and create an index of:

E(w) =   
[0, if "the"  
1, if "dog"  
2, if "is"  
3, if "brown"  
4, if "black"]

Outside of turning strings into numerical values, vectorization also optimizes data processing. Instead of having to go through identical strings several times, the same index is used akin to compressing files.

Finally, TFIDF (term frequency-inverse document frequency) is one of the ways to weigh term importance across documents. In simple terms, it takes each term, assesses its frequency divided by the document length, and assigns a weighted value to it. As a result, words that repeat frequently are considered more important.

**Line 3**

*LogisticRegression* is one of the ways to discover relationships between variables. Since our task is a classic example of classification, logistic regressions work perfectly as they take some input variable *x* (keyword)and assign it a value of *y* (informational/transactional/navigational).

There are other options, such as *LinearSVC,* which involves significantly more complicated mathematics. In extremely simplistic terms, SVC takes several clusters of data points and finds the values in each that are closest to the opposing cluster(s). These are called support vectors.

A hyperplane (i.e., an *n-dimensional* geometrical object in an *n+1-dimensional* space) is drawn in such a way that the distances between it and each support vector is maximized.

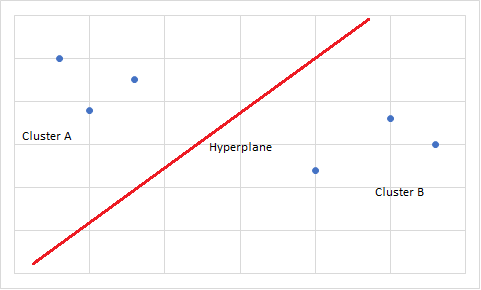


Image by author

[There exists research to state that using Support Vector Machines](https://link.springer.com/chapter/10.1007/BFb0026683) might produce better results in text classification, however, it’s likely due to the significantly more complicated nature of the task. These advantages aren’t entirely relevant in our case as they surface when feature counts reach inordinately high numbers, so linear regressions should work just fine.

**Line 4**

*Pipeline* is a flexible machine learning tool that lets you create an object that assembles several steps of the entire process into one. It has numerous benefits — from helping you write neater code to preventing data leakage.

**Line 5**

While not entirely necessary in our case, *SelectKBest* and *chi2* help optimize models by improving accuracy and reducing training time. *SelectKBest* allows us to set a maximum number of features that are used.

*Chi2* (or *chi-squared*) is a statistical test for the independence of variables that helps us select the best features (hence, *SelectKBest)* for training:

Chi squared formula.

Image by author.

where: c is degrees of freedom (i.e. sample size minus one)  
O is the observed value(s)  
E is the expected value(s)

Expected values are calculated by accepting the null hypothesis (variables are independent). These are then hedged against our observed values. If observed values deviate a significant margin from the expected ones, we can reject the null hypothesis, which forces us to accept that variables are dependent.

If variables are dependent, they are acceptable for the machine learning model as that’s exactly what we’re looking for — relations between objects. In turn, *SelectKBest* takes all *chi2* results and selects those that have the strongest relationships.

In our case, since our number of features is relatively small, *SelectKBest* might not bring the optimization we’d be interested in, but it becomes essential once the numbers start rising.

**Line 6**

Our final import is from NLTK, which we will only use for the *stopwords* list. Unfortunately, the default list isn’t suitable for our task at hand. Most such lists include words like “how,” “what,” “why,” and many others that, while useless in regular categorization, indicate search intent.

In fact, there’s a case to be made that these words are more important than any remainder in keywords like “how to build a web scraper.” Since we’re interested in the category of the sentence rather than any other value, the *stopwords* create the best shot at deciding what it might be.

As such, removing some of the entries from the stopwords list is vital. Luckily, NLTK stopwords are just text files which you can edit with any word processor.

[NLTK downloads are stored in user directories by default](https://sites.pitt.edu/~naraehan/python3/faq.html#Q-where-nltk-data) but can be changed if necessary through the use of *download\_dir=.*

**Dataframes and stopwords**

All machine learning models begin with data preparation and processing. Since we’re working with SEO keywords, these can be easily exported through CSV from popular tools that measure performance.

There is something to be said about picking a random sample that should include close to equal amounts of our categories. As we’re producing a pre-MVP, that shouldn’t be a concern, as data can be added as we go if the model delivers the results we need.

Before proceeding onwards, it would be wise to select a few dozen keywords out of a CSV file and label them. Once we get to a working model, we can label the rest. Since *Pandas* creates data frames in a tabular format, the easiest way is to simply add a new column, “Category” or “Label,” and assign each keyword row with *Informational, Transactional, or Navigational.*

df = pd.read\_csv('[KEYWORD\_LIST].csv')  
data = pd.DataFrame(df)  
  
words = stopwords.words('english\_adjusted')

**Line 1 & 2**

Whenever we have a CSV of any sort, *Pandas* requires us to create a dataframe. First, we’ll read the keyword list supplied by SEO tools. Remember that the CSV files should already have some keyword categorization involved, otherwise there will be nothing to train the model on.

After reading the file, we create a dataframe object from our CSV.

**Line 3**

We’ll use NLTK to grab the stopwords file, however, we can’t use it as it is. NLTK’s default includes many words we consider essential for keyword categorization (e.g., “what,” “how,” “where,” etc.). As such, it will have to be adjusted to fit our purposes.

While there are no hard and fast rules in such a case, indefinite and definite articles can stay (e.g., “a,” “an,” “the,” etc.) as they provide no information. Everything that could potentially show user intention, however, will have to be removed from the default file.

I created a copy called ‘english\_adjusted’ to make things easier for myself. Additionally, in case I need the original version for whatever reason, it will always be available without redownload.

Finally, you’ll likely need to run NLTK once with the regular parameter ‘english’ to download the files, which can be done at any stage. Otherwise, you’ll receive an error.

**Setting up the pipeline**

After all of these preparatory steps, we finally get to move on to the actual machine learning bit. These are the most important bits and pieces of the model. It’s likely that you’ll spend quite a bit of time tinkering with these parameters to find out the best options.

Unfortunately, there isn’t a lot of guidance that would apply in all cases. Some experimentation and reasoning will be required to reduce the amount of testing that’s needed, but eliminating it completely is impossible.

pipeline = Pipeline([('vect', TfidfVectorizer(ngram\_range=(1, 3), stop\_words=words)),  
 ('chi', SelectKBest(chi2, k='all')),  
 ('clf', LogisticRegression(C=1.0, penalty='l2', max\_iter=1000, dual=False))])

Some may notice that I’m not splitting the dataset into a train and test split through *scikit-learn*. Again, that is a luxury awarded by the nature of the problem. SEO tools can export thousands of (unlabeled) keywords in less than a minute, meaning you can procure a test set separately without much effort.

So, due to optimization reasons, I’ll be simply using a second dataset that has no labels as our testing grounds. Since, however, the *train\_test\_split* is so ubiquitous, I’ll show a version of the same model using it in the addendum at the bottom of the article.

**Line 1**

Pipeline allows us to truncate and simplify long processes into a single object, making it a lot easier to work with the settings of the model. It will also reduce the likelihood of making errors.

We’ll start by defining our vectorizer. I’ve noted above that we’ll be using *TFIDFVectorizer* as it produces better results due to the way it weighs words found in documents. *CountVectorizer* is an option, however, you’d have to import it, and the results may vary.

*Ngram\_range* is an interesting reasoning challenge. To get the best results, you have to decide how many tokens (in our case, words) have to be counted. *Ngram\_range* of (1, 1) would take a single word (unigram), of (1, 2) would take both a single word and the two nearest (bigram) in combination, of (1, 3) would take a single word, two, and three (trigram) in combination.

I chose *ngram\_range(1, 3)* for several reasons. First, since the model is relatively simple and performance is not an issue, I can afford to run a larger range of ngrams, so the lower bound can be set to be minimal.

On the other hand, once we remove stopwords, we should think about what ngram upper end would be enough to glean meaning from the keywords. If possible, I find it easier to pick the hardest and easiest examples out of the dataset. In our case, the easiest examples are any question (“how to get proxies”), and the hardest are nouns (“web scraper”) or names (“Oxylabs”)

Since we’ll be removing words like “to”, we get a trigram in question cases (“how get proxies”), which is completely clear. In fact, you could make the argument that a bigram (“how get”) is enough as the intention is still clear.

Hardest examples, however, will usually be shorter than a trigram as the ease of understanding search intent correlates with query length. Therefore, *ngram\_range (1, 3)* should strike a decent balance for performance and accuracy.

Finally, there’s an argument to be made for *sublinear\_tf*, which is a modification of the regular TF-IDF calculations. If set to *True,* weight is calculated through a logarithmic function: *1 + log(tf)*. In other words, term frequency gains diminishing returns.

With *sublinear\_tf,* words that appear frequently and in many documents would not be weighed as heavily. Since we have a collection of somewhat random keywords, we never know which ones get preferential treatment, however, these could often be terms such as “how,” “what,” etc., which are ones we’d like to be weighed heavily.

Throughout testing, I found that the model performed better without *sublinear\_tf*, but I recommend tinkering a bit to see whether it would grant any benefits.

The *Stopwords* parameter is, by now, self explanatory as we’ve discussed previously.

**Line 2**

While not technically a new line, I’ll be separating these out for clarity and brevity purposes. We’ll be now invoking *SelectKBest*, which I’ve written fairly extensively about above. Our point of interest is the *k* value.

These will be different, depending on the size of your dataset. *SelectKBest* is intended to optimize performance and accuracy. In my case, sending in ‘all’ works, but you’ll usually have to pick some large enough *N* that matches your own dataset.

**Line 3**

Finally, we get to the method that will be used for the model. *LogisticRegression* is our choice, as mentioned previously, but there’s a lot of tinkering to be done with the parameters.

“C”value is a hyperparameter, which is a parameter that tells the model the parameters it should pick. Hyperparameters are highly complicated parts of the model that have a tremendous impact on the end results.

In extremely simple terms, the *C* value is the trust score for your training data. A high *C* value means that a higher weight, when fitting, will be placed on training data and a lower weight on penalties. Low C values place higher emphasis on penalties and lower weight on training data.

There should always be some penalty in place as training will never fully represent real world values (due to being a small subset of it, regardless of how much you collect). Additionally, having outliers and not penalizing them means the model [will inch closer to being overfit](https://medium.com/p/7aeef64755d2).

The *penalty* parameter is the operation that will be used for the hyperparameter. There are three types of penalties offered by *SciKit-Learn* — *‘l1’*, *‘l2’*, and *‘elasticnet’*. ‘*None*’is also an option, but it should be used sparingly, if ever.

‘*L1*’ is the absolute sum of the magnitude of all coefficients. In simple terms, it pulls all coefficients towards some central point. If large penalties are applied, some data points can become zero (i.e., be eliminated).

‘*L1*’ should be used in cases where there is either multicollinearity (several variables are correlated) or when you want to simplify the model. Since *l1* eliminates some data points, models nearly always become simpler. It doesn’t work as well, however, when you already have a relatively simple distribution of data points.

‘*L2*’ is a different version of a similar process. Instead of being the absolute sum, it’s the sum of the square of all coefficient values. As such, all coefficients are shrunk by an identical value, but none are eliminated. ‘*L2*’ is the default setting as it’s the most flexible and rarely causes issues.

‘*Elasticnet*’ is a combination of both of the above methods. There [has been quite an extensive commentary](https://stats.stackexchange.com/questions/184029/what-is-elastic-net-regularization-and-how-does-it-solve-the-drawbacks-of-ridge) written on whether ‘*elasticnet*’ should be the default approach, however, not all solvers support it. In our case, we’d need to switch to the “saga” solver, which is intended for large datasets.

There would likely be little benefit to using ‘*elasticnet*’ in a tutorial-level machine learning model. Just keep in mind that it may be beneficial in the future.

Moving on to *‘max\_iter*’, the parameter will set the maximum number of iterations the model will perform until convergence. In simple terms, convergence is the point at which further iterations are unlikely to occur and serves as the stopping point.

Higher values increase computational complexity but may result in better overall behavior. In cases where the datasets are relatively simplistic, *‘max\_iter’* can be set to thousands and above as it won’t be too taxing on the system.

If the values are too low and convergence fails, a warning message will be displayed. As such, it’s not that difficult to find the lowest possible value and to work up from there.

**Fitting the model and outputting data**

We’re nearing the end of the tutorial as we finally get to fitting the model and receiving the output.

model = pipeline.fit(data.Keyword, data.Type)  
chi = model.named\_steps['chi']  
clf = model.named\_steps['clf']  
  
doutput = pd.read\_csv('[TEST\_KEYWORD\_LIST].csv')  
  
doutput['Type'] = model.predict(doutput['Keyword'])  
  
doutput.to\_csv('[RESULT\_LIST].csv')  
##print('Accuracy score ' + str(model.score(x\_test, y\_test)))

**Line 1–3**

Within line 1, we use our established pipeline to fit the model to the training data. In case some debugging or additional analysis is needed, the pipeline enables us to create named steps, which can be called later on.

**Line 4–8**

We create another dataframe from a CSV file that holds only the keywords. We will be using our newly created model to predict each keyword and its category.

Since our dataframe contains only keywords, we add a new column “type” and run *model.predict* to provide us with an output.

Finally, all of it is moved to an output CSV file, which will be created in the local directory. Usually, you’d like to set some destination, but for testing purposes, there’s often no need to do so.

There’s a commented-out line that I’d like to mention that calls the *score* function. *SciKit* provides us with numerous ways to estimate the predictive power of our model. These shouldn’t be understood as gospel, as predicted accuracy and real world accuracy can often diverge.

Scores, however, are useful as a rule of thumb and as a quick way to evaluate whether the parameters have had some influence on the model. While there are plenty of scoring methods, the basic *model.score* uses *R squared*, which is helpful in most cases whenever we’re tuning parameters.

**Examining the results**

My training data had a mere 1300 entries with three distinct categories, which I have mentioned above. Even with such a small set, the model managed to arrive at a decent accuracy score of about 80%.

Some of these, as one would expect, are debatable, and even Google thinks so. For example, “web scrape” was a keyword frequently searched for. There’s no clear indication of whether the query is transactional or informational. Google SERPs think as much as there are results for products and informational articles in the top 5.

There’s one area the model struggled with — navigational keywords. If I were to guess, the model predicted the category correctly about 5–10% of the time. There are several reasons for such an occurrence.

The distribution of the dataset could be blamed as it’s heavily imbalanced:

* **Transactional** — 0.457353%
* **Informational** — 0.450735%
* **Navigational** — 0.091912%

While real world scenarios would present a similar distribution (due to the inherent rarity of Navigational keywords), the training data is too sparse for proper fitting. Additionally, the frequency of navigational keywords is so low that the model would produce greater accuracy by always assigning the other two.

I don’t think, however, that presenting the training data with more navigational keywords would produce a much better result. It’s a problem that is extremely difficult to solve through textual analysis, whatever kind we choose.

Navigational keywords consist mostly of brand names, which are neologisms or other newly produced words. Nothing within them follows the natural language, and, as such, connections between them can only be discovered *a posteriori*. In other words, we’d have to first know it’s a brand name, from other data sources, to assign the category correctly.

If I had to guess, Google and other search engines discover brand names through the way users act when they query a new word. They might look for domain matches or other data, but predicting that something is a navigational keyword without human interaction is extremely difficult.

Feature engineering would be a potential solution to the problem. We’d have to discover new connections between the navigational and other categories and implement assignments through other approaches.

As feature engineering is a different topic entirely and one that deserves its own article, I’ll provide a single example. Navigational keywords will rarely be queried as questions (outside of “what is”) as they would otherwise make no sense (e.g., “how to Oxylabs,” “how to get Oxylabs.”)

There’s a debatable point as to whether “how to get Oxylabs proxies” would be considered transactional or navigational. It would definitely fit within the transactional category, however, so it could be considered so.

By knowing that relatively few navigational keywords would be formed as questions, we could build a model that would filter out most questions, leaving us with a smaller subset of potential targets.

Additionally, many navigational keywords have significantly shorter query lengths, mostly consisting of a single word, while the other categories have the same length relatively rarely.

Both of these methods and many others can be combined to improve the model’s accuracy when selecting navigational keywords. Getting into feature engineering, however, is much more complicated than a basic tutorial should cover.

For now, word classification should be covered with an overall better understanding of how machine learning models work. Hopefully, the explanation of the many parameters and tools available will let you create a functional model from the get-go.

**Conclusion**

Even if the article has been exceedingly long, you might have noticed that writing a machine learning model isn’t all that difficult. In fact, one may say that doing so is the smallest part of the project, at least in this case.

Machine learning heavily relies on preparation, of which we can outline several parts:

1. **Picking the right problem**. Some problems are simply better solved with other approaches. Don’t buy into the hype and try to solve everything through machine learning. With rule-based systems, you might be able to save time and resources while producing even better results.
2. **Preparing the data**. A model will only be as good as the data. If your data is labeled incorrectly, lacks veracity, or is otherwise faulty, no amount of development and resources build something that creates reliable outputs.
3. **Picking the model**. It’s easy to default to logistic regression or any other model because you’ve done it so many times. *Sci-Kit Learn* has other options such as those I haven’t even mentioned, such as *PassiveAggressiveClassifier,* which use different mathematical approaches. Again, I stress the importance of picking the right problem as it should decide what modeling method you choose.

I hope that this article will serve many newcomers to machine learning by providing not only the practical part, but also provide the way of thinking one should approach problems with.

**Addendum: Original full code block**

import pandas as pd  
from sklearn.feature\_extraction.text import TfidfVectorizer  
from nltk.corpus import stopwords  
from sklearn.linear\_model import LogisticRegression  
from sklearn.pipeline import Pipeline  
from sklearn.feature\_selection import SelectKBest, chi2  
  
df = pd.read\_csv('[KEYWORD\_LIST].csv')  
data = pd.DataFrame(df)  
  
words = stopwords.words('english\_adjusted')  
  
pipeline = Pipeline([('vect', TfidfVectorizer(ngram\_range=(1, 3), stop\_words=words)),  
 ('chi', SelectKBest(chi2, k='all')),  
 ('clf', LogisticRegression(C=1.0, penalty='l2', max\_iter=1000))])  
  
model = pipeline.fit(data.Keyword, data.Type)  
chi = model.named\_steps['chi']  
clf = model.named\_steps['clf']  
  
doutput = pd.read\_csv('[TEST\_KEYWORD\_LIST].csv')  
  
doutput['Type'] = model.predict(doutput['Keyword'])  
  
doutput.to\_csv('[RESULT\_LIST].csv')  
##print('accuracy score ' + str(model.score(x\_test, y\_test)))

**Addendum II: Train\_test\_split**

**Imports**

import pandas as pd  
from sklearn.feature\_extraction.text import TfidfVectorizer  
from sklearn.linear\_model import LogisticRegression  
from sklearn.pipeline import Pipeline  
from sklearn.model\_selection import train\_test\_split  
from sklearn.feature\_selection import SelectKBest, chi2  
from nltk.corpus import stopwords

As per usual, we have to import the *train\_test\_split* itself (line 5).

**Setting up the split**

x\_train, x\_test, y\_train, y\_test = train\_test\_split(data.Keyword, data.Type, test\_size=0.3)  
  
pipeline = Pipeline([('vect', TfidfVectorizer(ngram\_range=(1, 3), stop\_words=words)),  
 ('chi', SelectKBest(chi2, k='all')),  
 ('clf', LogisticRegression(C=1.0, penalty='l2', max\_iter=1000, dual=False))])

**Line 1**

As our dataset has only two features (*keyword* & *category*), we’ll need two variables for each. One of them will store the training data, and the other one will be used for testing purposes.

We’ll take the data frame created in previous steps and assign the column names (in my dataset, they were called “Keyword” and “Type,” as evidenced by the parameters).

Finally, *SciKit-Learn* solves data splitting problems for us by allowing automated divisions for both sets. The *train\_test\_split* takes float and integer values that represent percentages to be used for either the test set size or the training set size. If both are left as *None*, it will default to 0.25.

Some tinkering will be required to get the best results. I’ve tried many different splits, with 0.3 producing the best results. Generally, you’ll find that many models will work best on splits ranging from 0.2 to 0.3.

Particular splits tend to have less effect on accuracy when data point numbers increase. In fact, in extremely large datasets, splitting on 0.1 might improve computational performance.

Relations between statistical units are complicated, however, the abstract field of connections that can be made is finite, so that accuracy can be understood as a requirement for a flat number of data points instead of a specific ratio. In other words, there is some *N* where results don’t get any better, so if datasets are large, smaller ratios might be more optimal.

There are some [highly technical articles written about the topic](https://link.springer.com/article/10.1007/s41664-018-0068-2) that explain the idea in much greater depth and provide ways to calculate the optimal split.

Everything else in this code block follows the same steps as in the original tutorial.

**Fitting the model and outputting data (again)**

model = pipeline.fit(x\_train, y\_train)  
chi = model.named\_steps['chi']  
clf = model.named\_steps['clf']  
  
doutput = pd.DataFrame({'Keyword': x\_test, 'Type': model.predict(x\_test)})  
  
doutput.to\_csv('[RESULT\_LIST].csv')  
##print('Accuracy score ' + str(model.score(x\_test, y\_test)))

**Line 1**

Instead of training our model on the labeled dataset directly, we’ll be training it on the previously split one, naming *x\_train* and *y\_train*. Lines 2 and 3 remain identical.

**Line 4**

Since there is no separate dataset, we’ll be using the test part of our initial one for predictions. So, we create a dataframe that has the column *Keyword* in which we will output all of the keywords from the test dataset. In the second column, *Type*, we’ll use the model to predict the category of the keyword, drawing from the same dataset.

Finally, as per the original version, all of that will be outputted into a results file. Printing accuracy score is also an option if one is interested in how well the model thinks it’s performing.