CIS575: Clustering Documents With Unsupervised Machine Learning

Table of Contents

- Watch: Embed Sentences Into Vectors
- Code: Practice Embedding Sentences Into Vectors
- Watch: Compute Similarity of Sentence Embeddings To Find Similar Movies
- Code: Practice Computing Similarity of Sentence Embeddings To Find Similar Movies
- Watch: Methods for Clustering
- Watch: Hierarchical Clustering of Movies Based on Their Descriptions
- Watch: Practice Hierarchical Clustering of Movies Based on Their Descriptions
- Watch: The Anatomy of a Dendrogram
- Quiz: Interpreting a Dendrogram
- Watch: Build Dendrograms Using Different Linkages
- Code: Practice Building Dendrograms Using Different Linkages
- Quiz: Identify Linkages in Dendrograms
- Watch: Measure Clustering Performance
- Code: Practice Measuring Clustering Performance
- Assignment: Course Project, Part Two — Performing Hierarchical Clustering on Sentence Embeddings To Group Similar Texts
- Module Wrap-up: Perform Hierarchical Clustering on Sentence Embeddings To Group Similar Texts

Module Introduction: Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts

- Watch: Calculate Centroids and Medoids To Find a Representative Data Point
- Watch: Practice Calculating Centroids and Medoids To Find a Representative Data Point
- Watch: Compute the Medoid To Find a Representative Movie
- Code: Practice Computing the Medoid To Find a Representative Movie
- Watch: Clustering Movies With K-Means and Evaluating Performance

- Code: Practice Clustering Movies With K-Means and Evaluating Performance
- Assignment: Course Project, Part Three — Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts
- Module Wrap-up: Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts

Welcome to Your Course

Video Transcript

Unsupervised learning methods allow us to organize documents by their similarity. This would be expensive to do manually for billions of documents. So an automation is a highly valuable cost reduction. But its implementation still requires human oversight and subjective judgment.

At the very least, an expert needs to decide on what it means for two or more documents to be similar. A computer cannot do this. But once a human chooses a similarity metric, the algorithm can automate and scale up that decision to a massive corpus of documents.

For example, millions of YouTube videos, with transcripts, are added daily. Often, they lack categorization or description. An expert can evaluate existing videos and decide on representative categories, say, engineering, education, or drone flight. Any new video transcript would be then compared, via string or numeric representation, to each existing category. And the decision will be made on which category that new video belongs to.

We start this course with several metrics that measure string similarity, and then advanced to two popular clustering algorithms, hierarchical clustering and k-means. We will also learn how to measure the quality of the identified clusters of documents and how to tune hyperparameters of these models. Let's get started.

Course Description

In the previous course, you calculated similarity scores on different documents to discover if different texts were related topically to one another. In this course, you will turn your attention to measuring distance. This can be considered the opposite of measuring similarity, as you are measuring the dissimilarity of the documents, expressed in how far apart they appear from one another in a visualization. The goal, however, is still to discover how alike or unlike various groups of text documents are to one another.

At scale, this is a problem you can encounter if you need to group thousands of products together purely by using their product description, or to make a recommendation: if you liked this particular movie, then you might also like this other movie. In this course, you will work with several different data sets and use both hierarchical and k-means clustering to create clusters, and you will practice with several distance measures to analyze document similarity. Finally, you will create visualizations that help to convey similarity in powerful ways, so stakeholders can easily understand the key takeaways of any clustering and distance measure that you create.

System requirements: This course contains a virtual programming environment that does not support the use of Safari, tablets, or mobile devices. Please use Chrome, Firefox, or Edge for this course. Refer to eCornell's **Technical Requirements** for up-to-date information on supported browsers.

What you'll do

- Use similarity and distance metrics to determine text similarity
- Use and evaluate hierarchical clustering to group similar texts
- Implement the k-means clustering algorithm to group similar texts and measure cluster quality

Faculty Author

Oleg Melnikov, Ph.D.

Visiting Lecturer

Oleg Melnikov received his Ph.D. in Statistics from Rice University, advised by Dr. Katherine Ensor on the thesis topic of non-negative matrix factorization (NMF) applied to time series. He currently leads a Data Science team at ShareThis Inc, Palo Alto, CA, and has decades of experience in mathematical and statistical research, teaching, databases and software development, finance (portfolio management and security analysis), AdTech and in other fields. His academic path started with a BS in Computer Science (and some years in pre-med). Now, he holds masters' degrees in CS (Machine Learning track) from Georgia Institute of Technology, Mathematics from UC-Irvine (where he was in Math Ph.D. program), Statistics from Rice University, MBA from UCLA, and Quantitative Finance

Cornell Bowers Computing and Information Science

Cornell University

(MFE equivalent). Oleg is passionate about education and hard sciences. He has been teaching statistics, machine learning, data science, quantitative finance, and programming courses at eCornell, Stanford University, UC Berkeley, Rice University, and UC-Irvine.

Read: Using Python and Jupyter Notebooks in This Course

Throughout the course, there will be opportunities for you to implement natural language processing (NLP) concepts from Professor Melnikov's videos. Python is the programming language used in this course, and you will learn how to perform NLP-related tasks using several of its popular libraries and tools.

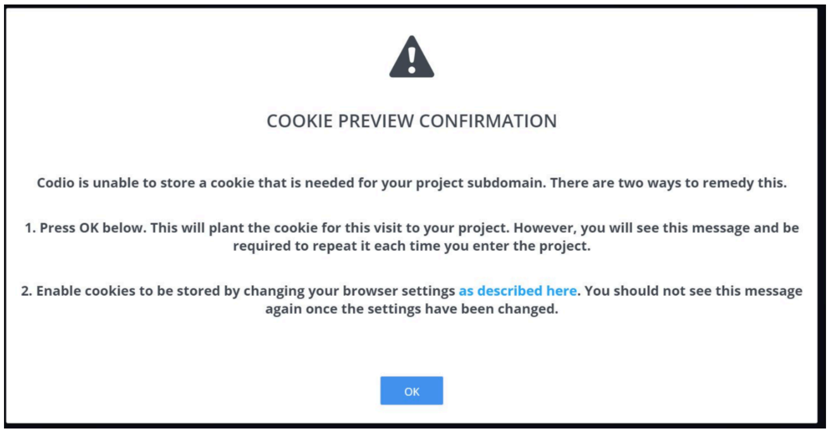
You will interact with Python throughout this course by writing and running code in workspaces hosted on Jupyter Notebooks. Our cloud-based implementation of Python and Jupyter Notebooks means that you can complete this course without installing Python on your machine.

About Your Workspace

Each workspace in which you write Python code is distinct from every other workspace in the course; this means that the code you develop on one page of the course *will not carry over* to another page. Each workspace *is* persistent, however, so if you save some work and log out of the course, you can return to that page later and pick up where you left off in your previous session.

If you need more room in which to work, you can make your workspace fullscreen by clicking the fullscreen button in the bottom-right corner of the workspace. The image below is an example of what the button looks like.

Sometimes pop-up messages can occur when you work in this workspace. The below pop-up notifies you of a cookie requirement by the website that hosts Jupyter Notebooks, documents in which you will write code.


COOKIE PREVIEW CONFIRMATION

Codio is unable to store a cookie that is needed for your project subdomain. There are two ways to remedy this.

1. Press OK below. This will plant the cookie for this visit to your project. However, you will see this message and be required to repeat it each time you enter the project.

2. Enable cookies to be stored by changing your browser settings as described here. You should not see this message again once the settings have been changed.

Follow the instructions to resolve this issue and continue.

Jupyter Notebooks in This Course

The activities and projects use Jupyter Notebooks, which are web-based interactive documents that can weave code, data visualizations, outputs, equations, and text together in a single document. Each notebook within the course is hosted on a separate, cloud-based virtual machine (a.k.a. Codio Unit) that has Python and the necessary packages/dependencies already pre-installed.

The notebook activities throughout the course have been structured for you to:

1. **Setup** your workspace by importing necessary Python libraries and data.
2. **Review** coding snippets and concepts demonstrated in the videos.
3. **Practice** tackling related tasks, which are ungraded.

At the end of each module, you will demonstrate the skills that you have acquired in the module by completing one part of the graded **Course Project**.

Best Practices

- Do all of your work in the Jupyter Notebook workspace.
- Comment code to explain what is happening or demonstrate that you understand what is happening, and cite any sources other than documentation and class material used to assist in an exercise.
- Create test cells and perform tests outside of the graded function block, and leave this content in place to show your progression and process.

Back to Table of Contents

Tool: Writing and Navigating Jupyter Notebooks

Throughout this course and certificate, you will write and run Python scripts via Jupyter Notebooks. Use the following tool to familiarize yourself with Jupyter Notebooks or to refresh your memory for how they function. You can also save it for future reference.

Please complete this activity in the course.

Back to Table of Contents

Read: Preview Your Course Project

Throughout this course, you will practice applying the teachings of each module in a multi-part course project. The final project part, in which you will synthesize all course findings, is worth a significant percentage of the whole. You can review all relevant information on the **Grades** page of this course.

☆ **Key Points**

You will apply the content from this course in a multi-part project.

Since each module builds on the previous ones, it is essential that you work through the course in the order it appears.

Note: Though your work will only be seen by eCornell, you should take care to obscure any information you feel might be of a sensitive or confidential nature.

Course Project Parts

Preview the course project below. As you work through the course, reflect on how this relates to the course content and to your experience.

– **Course Project, Part One — Using Metrics To Determine Text Similarity**

In this first part of the course project, you will be writing a function to compute Jaccard similarity scores for presidential inaugural speeches and calculating the Hamming distance between two strings of varying lengths.

– **Course Project, Part Two — Performing Hierarchical Clustering on Sentence Embeddings To Group Similar Texts**

In this part of the course project, you will be encoding movie descriptions into numeric vectors in order to cluster them with various sets of parameters and determining the best set of parameters for hierarchical clustering using the silhouette score.

– **Course Project, Part Three — Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts**

For the final part of the course project, you will be encoding movie descriptions into numeric vectors in order to cluster them with various sets of parameters and determining the best set of parameters for k-means clustering using the silhouette score.

Back to Table of Contents

Discussion: Project Forum

Post in this project forum if you need more information about or help with the course project. The course facilitator will monitor this discussion at least once per day.

To participate in this discussion:

- Add a URL link to the page of concern in your post and briefly explain what you are trying to achieve, what you have tried, and where you are stuck
- Do **not** include your solution code (whether or not it is correct).
- Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding **plagiarism** (the presentation of someone else's work as your own without source credit).

Important Note Regarding the Sharing of Python Code

Since this course includes code-based exercises and project parts, we ask that you refrain from posting solution code to this discussion. You are encouraged to help each other as you encounter challenges along the way, but please offer guidance and suggestions rather than solutions. You may post snippets of code when asking for or providing help by highlighting your code text in your reply and clicking **Format > Code**. However, do **not** post solution code (whether you think it is correct or not). The goal is to assist each other, not share answers. Please reach out to your course facilitator with any questions.

Back to Table of Contents

Quiz: Datasets Disclaimer

Before continuing with this course, please read the statement below and acknowledge your understanding by selecting "I understand."

Please complete this activity in the course.

Back to Table of Contents

Module Introduction: Use Metrics To Determine Text Similarity

Computers may not be able to categorize books or movies as intuitively as humans can, but once given a standard by which to measure similarity, they will do so much more efficiently. However, before you begin categorizing entire documents, you will first practice identifying differences in shorter strings, such as words.

In this module, you will first become familiar with two major similarity measures: lexical and semantic similarity. Then, you will practice evaluating the similarity between words by computing the distance between them and approach virus identification and spell-check problems using the Hamming and Levenshtein distance metrics. Measuring the similarity between words in this module will prepare you for the document categorization later in the course.

Back to Table of Contents

Watch: Compare Texts Using Similarity Metrics

Lexical and semantic similarity are two measures you can use to evaluate the distance between words or documents. *Lexical similarity*, which is based on syntax, structure, and content, is commonly used for autocomplete and spell check applications. *Semantic similarity*, by contrast, uses context to evaluate the similarity in meaning between words or documents.

Libraries/Methods

Standard Library

`set.intersection()`, `len()`
`set.union()`, `ord()`, `corr()`
`np.sum()`, `zip()`, `len()`
`random.choice()`
`pd.DataFrame()`

In this video, Professor Melnikov demonstrates how to compare two words using lexical similarity measures in code. First, he quantifies the distance between words in terms of similar letters by calculating the Jaccard similarity. Then, he computes word similarity using a correlation function.

Note: Consider watching these videos in full-screen mode so you can see the Python commands clearly.

Video Transcript

Similarity measures estimate the proximity between entities. Specifically in term similarity, entities are words. And in a document similarity, entities are documents. The proximity also needs to be defined a bit more precisely.

In lexical similarity, we relate entities on the basis of syntax, structure, and content of the text. In semantic similarity, we relate entities on the basis of semantic. Meaning census and context of the text. We have already computed semantic similarity when we represented words, phrases, and sentences as numeric vectors, and computed there are dot products, Euclidean distances, and cosine similarities.

So in this module, we cover lexical similarity, which is prominently applied in order completers, spell checkers, and spell characters. Let's see lexical term similarity in code. We start with the foundational similarity. And that's comparing two elements regardless of their structure, whether they're exactly the same or not. This is a binary or Boolean comparison, where true or false is returned, or 01 represented by false or true.

We can use that in building operations on offsets. In particular, intersection compares all the elements and keeps only those from the two sets that are matching exactly. And union keeps unique instance from both sets. The word Colonel and E Colonel are presented as sets of characters here. Notice that neither intersection, or the union, will have the characters in the order. The sets do not remember the order. And they present each character only once. So the letter L and the letter E is captured only once.

We can use this operations to build a Jacquard similarity. Jacquard similarity is a value between 0 and 1 computed from the size of the intersection or the cardinality of the intersection divided by the cardinality of the union, or the size, length, or number of elements in the set from the union operation. We can have this Jacquard demo where we are comparing Jacquard similarity for different words. Notice that if the character is repeated multiple times, it's not accounted. So Cornell and E Cornell has a Jacquard similarity of 1.

And we can further evaluate characters as they're unicode numbers. So each character can be represented as a number with the function Ord, which converts characters to numbers. And in this vectorized terms function, we are processing more person every term into characters replacing the characters with the character codes. We're keeping the order. And if the particular term is too short, has too few characters, it is padded with Nance, or not a number value, in the resulting data frame.

This allows us to now treat this Ward sequence of numbers as vectors and compare their similarity with, let's say, a correlation function, which we use here. It's the easiest. So it's already built into the data frame. So we compute the correlation.

Before we do that, we have to replace all NAM values with 0. Otherwise, We'll get lots of NAMs in this resulting correlation matrix, which is a square symmetric matrix with the once on diagonal representing the similarity between word vector to itself, and minus 1, which is the lowest value. It's the perfect negative correlation. And one being the perfect positive correlation.

So we can see that the words that have the same beginning have higher similarity, or even similar codes. Like the word cats and cows have high similarity of 0.997 because of the values that they use. And catfish and cat are different semantically, but they are still very similar because they start with the same cat.

Code: Practice Comparing Texts Using Similarity Metrics

Previously, Professor Melnikov demonstrated how to calculate Jaccard similarity and use a correlation function to compare two words. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice applying Jaccard similarity to both characters and words.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: Find Mismatches With Hamming Distance

Hamming distance is another way to measure the distance between words, when working with two strings of equal length. A lesser Hamming distance means the two strings you are comparing are fairly similar, while a greater Hamming distance indicates dissimilar strings. Two fields that often use Hamming distance include cryptography and computational biology.

Libraries/Methods

Standard Library

`set.intersection()`, `len()`
`set.union()`, `ord()`, `corr()`
`np.sum()`, `zip()`, `len()`
`random.choice()`
`pd.DataFrame()`

In this video, Professor Melnikov uses Hamming distance to measure the dissimilarity between DNA sequences. In this scenario, he identifies any viruses present in a patient sample by computing the Hamming distance between the sample's DNA and 10,000 known DNA virus sequences in a dataframe.

Note: Consider watching these videos in full-screen mode so you can see the Python commands clearly.

Video Transcript

Hamming distance is a string metric which counts element-wise mismatches in the two strings of equal length. In other words, it is the number of substitutions needed to make two strings equal. It is a simple and popular metric and often used in spell check, and cryptography, and binary code error correction or detection.

It is also applied in genetics and computational biology to measure the dissimilarity between the two DNA sequences. DNA molecule or sequence is a long chain of nucleotides often represented by characters A, C, G, T. It's not a natural text, but it is a string nonetheless, and it benefits from natural language processing techniques.

This string encoding of a biological structure is often full of inaccuracies due to DNA mutations and errors in the process of DNA sequencing, which is a process of DNA digitizing. We might be interested in identifying whether a DNA is drawn from a patient's saliva matches any of the DNA strings from the bank of viral DNAs.

Hamming distance is often used to produce the degree of dissimilarity from each pair of DNA strings. Let's see this in code. We are creating the function Hamming distance, and it will take two strings, S1 and S2. If their length is unequal, the Hamming distance will return infinite distance, which is a valued in NumPy.

Otherwise, it will count the number of mismatches character by character. For that, it zips the two strings into a generator of tuples of characters in corresponding positions. So we can easier match them.

Finally, we sum up all the truths and false. Remember, those are represented by zeros and ones or equivalent to zeros and ones. And in essence, we're counting number of truths, number of mismatches.

In an example, this is number three for the cat and dog because none of the characters in the corresponding positions match. The ACGT and ACCT counts one in the third position, and cats and dogs returns infinity for the Hamming distance because of the different length.

Gene sequence is a function that we can use to generate a sequence of nucleotides. And right now we're generating 500, but actual DNA sequences could have millions of nucleotides that need to be compared in a fast manner, which is where Hamming distance is very useful.

This will use NumPy random choice to sample with a replacement from the ACGT list of characters. And we can similarly generate a binary code. Remember that large files consisting of megabytes or gigabytes of code will have lots and lots of zeros and ones.

And we can use Hamming distance to compare those codes and find places where a particular stretches of zeros and ones have been corrupted by some virus. But that would be a virus in the coding domain, not biological domain.

Now we can compute the distance between two generated DNA sequences. So we have two of them generated with the size of 30. If they need to be the same size, DNA x is DNA we might be interested in identifying. DNA 1 here is a particular DNA that we are familiar with so known DNA 1.

And let's say we have a bank of these DNAs, 10,000 of them packed in a data frame. And we can take DNA x and compare or compute the Hamming distance of the DNA x against each one of the viral DNAs. Once we order all these viral DNAs by the distance to x, d to x column, we have the topmost few DNAs which have distance 13 to the DNA of concern, which is DNA x.

This is how we can find the virus most closely related to the virus retrieved from patients' saliva or some other sample.

Back to Table of Contents

Code: Practice Finding Mismatches With Hamming Distance

Previously, Professor Melnikov demonstrated how to calculate Jaccard similarity and measure the dissimilarity between DNA sequences using Hamming distance. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice applying Jaccard similarity and computing the hamming distance of all virus codes in a database to identify the query virus.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Similarity Measures

In this module, Professor Melnikov has introduced different types of similarities. Now it is time to test your understanding of them. Select your responses, keeping in mind that a question may have more than one correct answer. Submit your quiz when you have finished.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve the maximum score.

Please complete this activity in the course.

Back to Table of Contents

Watch: Comparing Sequences With Levenshtein Distance

Up to this point, you have practiced using two metrics to evaluate the distance between entities: Jaccard similarity and Hamming distance. A third metric, *Levenshtein distance***,** counts the number of alterations needed to convert one string into another string using addition, deletion, or substitution. You can use this metric to compare characters between two strings, two phrases, or one string and one phrase. This metric is often used to determine the quality of language translations or to detect a computer virus by checking binary code.

Here, Professor Melnikov demonstrates how to compute Levenshtein distance using dynamic programming, which solves a complex problem by breaking it into subproblems and solving them recursively. Follow along with this video, as he models the results in a 2D matrix and explains how to interpret them.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Levenstein or edit distance is another metric named after a Soviet mathematician Vladimir Levenshtein, who published this metric in 1965. It is minimal count of character additions, deletions, and substitutions needed to change one string into another. For example, change in Cornell to eCornell takes one edit operation of additions or insertion of a letter E. Change in cat to dog takes three substitution edits.

A typical algorithm for edit distance uses dynamic programming, or DP, where a complex problem is recursively solved as a series of smaller sub problems. In each sub problem, we take the best edit distance from the previous evaluation of two prefixes and keep it as is if the current characters match or add one and proceed with three edit operations.

This fills up a two-dimensional matrix of best edit distances from the top left to the bottom right of this matrix, where the rows and columns associated with the characters of each word. Let's see this algorithm in code.

We're defining a function at a distance, edit distance, with two strings passed into it and a show matrix if we want to display the results in dynamic programming matrix. We're flipping the strings to keep the first one as the shortest for convenience and initialize in the matrix M, as that contains the best edit distances to have all zeros. This is a list of lists, not necessarily an a NumPy array.

Then we iterate over each row. And within each row, we iterate over the columns. So we fill out the columns first before we move on to the next row, fill up the next row with going over the columns and so forth. If there is nothing-- if one string has ended, we just copy the rest of the second string.

If the current character ij-- if the previous character ij match, then the current character is copied over from the previous one. Alternatively, if none of these conditions are satisfied, we are looking at the minimal operation from the insertion, removal, and substitution and are-- from the previous operation and are adding 1 to that operation because the mismatched characters in this case would require one operation other-- either one of this.

Let's see how this looks in the matrix. So if we execute this, the word cat and the rats are two distances apart because we need to change c with r, and we need to prepend or delete S. And as we progress from the top left of this matrix where the two blank characters or empty characters match to the bottom right, we are indicating where the characters need to be edit or deleted.

And we are filling this matrix from the left to the right and top, bottom, row by row. The top, bottom corner of this matrix indicates the number of edits that need to be made. And we can trace back the path of operations from the bottom right to the top left if we need to recover all the operations that have been made.

Likewise, we can compare a single word to the full string, character by character. And similarly, we can apply the same edit distance operation to a set or sequence of words in the sentence, or two sentences. So we're not comparing characters anymore.

We're comparing words in these two phrases. And this is often useful in measuring the quality of translations, which can sometimes miss articles or modify a particular word. Another operation is to check the binary code to some ground truth code to locate the place of a virus or deleted part, computer virus, or deleted part of the code.

And one more prominent application is comparing different DNAs, which consists of nucleotide characters-- A, C, G, T-- and locating where the two DNAs may have been mutated or misread during the sequencing, DNA sequencing process.
Back to Table of Contents

Activity: Compute Levenshtein Distance

At its core, Levenshtein distance is a simple distance metric which might be summarized as the number of character changes required to make two text objects identical. In this ungraded activity, you will practice evaluating the Levenshtein distance between various phrases.

Evaluate the Levenshtein distance for each of the given sentences. Click **Solution** to check your response.

Example 1

- "The **cat** went for a walk."
- "The **dog** went for a walk."

Solution

These two sentences are identical except for the words “cat” and “dog.” To make them identical, you would need to make three changes. In this case, substitute “c” for “d”, “a” for “o”, and “t” for “g.” Thus, the Levenshtein Distance in this case would be three.

Example 2

- The **cat** went for a walk.
- The **hat** went for a walk.

Solution

In this example, “cat” and “hat” are the only differences between the two sentences. However, we only need to change the “c” to an “h” to make them identical, and so the Levenshtein Distance would be one.

Example 3

- The **cat** went for a walk.

- The **coat** went for a walk.

Solution

Here, the words “cat” and “coat” are the only differences. To make them identical, you would simply add an “o” as the second letter of “cat,” and thus the Levenshtein Distance here is once again one.

Example 4

- The cat went for a walk.
- The cat went for a walk **in the park**.

Solution

In this example, the first parts of the sentences are identical, but the second sentence has an extra phrase. Including the extra spaces, “ in the park” would be a total of 12 insertions to make the first sentence identical to the second, and so the Levenshtein Distance would be 12.

This is another example where choosing the right distance metric for a given job is important in Natural Language Processing. One might argue that semantically, those last two sentences are closer in meaning than any of the first three examples. However, because Levenshtein Distance only measures character changes, by this distance metric example four would be judged as the farthest distance apart. Thus, Levenshtein Distance might not be appropriate for NLP problems attempting to determine semantic similarity. However, if you were comparing two documents to try to detect plagiarism, Levenshtein Distance would be an excellent data point in uncovering suspicious activity.

There is no submission requirement for this ungraded activity.

Back to Table of Contents

Code: Practice Comparing Sequences With Levenshtein Distance

Previously, Professor Melnikov demonstrated how to compute Levenshtein distance using dynamic programming and how to model the results in a 2D matrix. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice computing the Levenshtein or Edit distance to find the difference between two direction paths and to find a similar name.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: Use Levenshtein Distance To Autocorrect Text

You probably experience at least one practical use of the Levenshtein distance metric every day whenever you use a spell checker. When correcting a misspelled word, spell checkers consider the best auto-correct option determined by the Levenshtein distance. However, when a misspelled word has many potential alternatives, methods calculating the Levenshtein distance can result in a long runtime for low-accuracy results.

Libraries/Methods

Standard Library

`python-levenshtein`
`nltk.edit_distance()`
`nltk.download()`, `timeit()`
`matplotlib.pylot`
`inspect.getsource()`

In this video, Professor Melnikov computes Levenshtein distance using three different functions: the built-in `Python-Levenshtein` package, the edit distance function he created in the previous activity, and the NLTK implementation of edit distance. He then uses the `timeit` function in conjunction with the Brown Corpus dictionary to offer recommendations for reducing the total number of computations and runtime.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

A common application of edit distance is spelling correction. To correct a misspelled word, we consider the closest candidates and pick the most likely candidate in the case a tie. As we know, some misspellings are more common than others because letters are mixed up on a computer keyboard, or due to a phonetic ambiguity, or other human and physical factors.

Since edit distance is discrete, the closest candidate can be at least one edit distance away. But this can produce too many candidates, especially for short words which allow for fewer letter permutations. For example, the word cat is one edit distance away from car, cap, cut, bat, cats, and 25 other candidates in most lexicons. So we would need more context to autocorrect a misspelled word cat.

On the other hand, the word interdisciplinary has no candidates within four edits, and only two candidates within seven edits. So it can be auto corrected with high confidence since most misspellings affect only one or two characters.

Some generalizations of edit distance allow weighted edits with weights drawn from the empirical observations of misspellings. This reduces the candidate pool and improves auto correction. Levenshtein distance is a sophisticated string metric, but it is notoriously slow, where its polynomial runtime is proportional to the product of length of the input words. Often we use heuristics and prior knowledge to avoid unnecessary edit distance computations in search of the needed candidates.

For example, we could rank and consider candidates which are within one character in length from the query word. We could also rely on assumption that misspellings of the first character are highly unlikely. This dramatically reduces the pool of candidates needing edit distance computation. Let's see some of these performance tricks in code.

While loading some of the packages, including the Python Levenshtein, which is a C language implementation of the Levenshtein distance. Then there is edit distance function that we created in Python. And also the NLTK's implementation is loaded. There's a magic function, Time It in Jupyter and Notebooks. And you can read the documentation by putting the question mark in front of it. But it allows us to time different lines of code and specify parameters on how many iterations would like it to execute.

Notice that the C implementations is hundreds of times faster than ours, or analytics implementation of the edit distance function. This NS is nanoseconds and MS is microseconds. You can also explore the edit distance function in NLTC by using the inspect, get source, which will print out the source code for that function.

Next, we're going to time the functions at different length of character inputs. Notice the quadratic increase in time, which is the vertical axis for NLTK and our implementation, and for levenshtein distance implementation. Although you cannot see the difference. It appears flat because it's hundreds of times faster. But if you are to show it on its own axis on the right-hand side, it will be just as curved upwards.

Next we're loading the brown corpus with a dictionary of 50,000 words, unique words. And we will experiment with some words that can possibly be misspelled. So cat has lots of candidates in brown corpus of one edit distance away, which makes correct since the word cat, or misspellings in it, very problematic.

If we are to paste surreptitious instead and query all the nearby words in the lexicon from brown corpus, we see that surreptitious can be corrected unambiguously with the first candidate. There is no ambiguity or there is no conflict or tie with the words that are close to it.

Likewise, we can do the same, but with fewer number of computations. Notice that previously, we did 50,000 edit distance computations, which took two plus seconds. We don't need to compute edit distances for the whole corpus because some of the words will be too short or too long. So we can only consider words that are one character shorter or longer. This is only 4,500, 4,600 computations of edit distances.

And if we are only looking at the words that start with the same character, assuming that it's unlikely to be misspelled, then it's only 500 edit distance computations that we need to make. And that's 45 milliseconds.

Back to Table of Contents

Code: Practice Using Levenshtein Distance To Autocorrect Text

Previously, Professor Melnikov demonstrated how to compute Levenshtein distance for practical applications using three different functions to determine the most efficient method. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice creating a function that will return all of the candidate words that are within a given distance of a query word, and computing the Edit distance between the query and your function results.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Tool: Review: Similarity and Distances Metrics

In this module, you have practiced evaluating the similarity between words or documents using various metrics. Some of these measures, such as Jaccard similarity, may be familiar to you. Others, such as the Levenshtein distance, may be new. Use this tool to review each metric and determine whether it measures similarity or distance.

Please complete this activity in the course.

Back to Table of Contents

Assignment: Course Project, Part One — Using Metrics To Determine Text Similarity

In Part One of the course project, you will be writing a function to compute Jaccard similarity scores for presidential inaugural speeches and calculating the Hamming distance between two strings of varying lengths. The tasks you will perform are based on videos and coding activities in the current module, but may also rely on your preparation in Python and basic math.

Resources

Use these resources to help you as you complete the course project:

- **Project Forum**
- **Jupyter Notebook Guide**

Use the tools from this module to help you complete this part of the course project. Additionally, you may consult your course facilitator and peers by posting in the project forum that is linked in the sidebar.

Completion of this project is a course requirement.

Instructions

Read the general project instructions within this accordion carefully, then follow the instructions specific to this part of the course project in the Jupyter Notebook below.

Please complete this activity in the course.

This exercise is **graded**, and may take ~3 hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Use Metrics To Determine Text Similarity

In this module, you gained a familiarity with several approaches and metrics used to measure the distance between words and documents. You began by comparing two words using Jaccard similarity, and then applied the Levenshtein and Hamming distance metrics to more complex problems. Finally, you built functions computing Jaccard similarity and Hamming distance, which can be used to evaluate the similarity between speeches and rank viral samples.

Back to Table of Contents

Module Introduction: Perform Hierarchical Clustering on Sentence Embeddings To Group Similar Texts

In this module, you will be introduced to *clustering*, an unsupervised method you will use to categorize documents based on their similarities. However, before you can practice clustering techniques in code, you will first examine how to represent documents as numerical vectors. Then, you will practice using *hierarchical clustering* to categorize movies in a dataset according to genre.

Finally, you will use *dendrograms*, or tree diagrams, to visually evaluate the quality of your clustering. You will also evaluate your clustering quality using several quantitative metrics.

Back to Table of Contents

Watch: Embed Sentences Into Vectors

When representing and evaluating many sentences in code, one helpful tool is **Sentence Bidirectional Encoder Representations from Transformers (SBERT)**, which encodes entire sentences or documents as numerical vectors. SBERT can be used for a variety of NLP tasks including language translation and meaningful sentence embeddings.

Libraries/Methods

Standard Library

`sort_values()`, `split()`
`scipy.spatial.distance.cosine()`

Sentence Transformers Library

`sbert.encode()`

Follow along with Professor Melnikov in this video, as he uses the `sentence-transformers` implementation of SBERT to encode 15 famous quotes. He then evaluates the quotes' correlation coefficients and cosine similarities using a pre-trained model.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Recently, a team at Google introduced a new way of encoding entire sentences or documents as numeric vectors. Their Bidirectional Encoder Representations from Transformers or BERT is a neural-network-based language model which can be extended for a dozen of key NLP tasks, including question and answering, language translation, and generation of meaningful sentence embeddings.

The UKP lab in Germany has extended BERT to Sentence BERT or SBERT which is incredibly easy to use and is only a few hundred megabytes compared to 8 gigabyte fastText model.

Sentence BERT does not store static word vectors. Instead each word vector, if still needed, is generated dynamically from its semantic context.

Let's try SBERT in code. Here we are loading the Sentence BERT. It's called Sentence Transformers Package. And importing this library into memory and loading the model, a pretrained model from UKP lab's website, which is one of many.

They are different on-- they are trained on different corporas. They have different language representations. Some of them are multilingual and can be more suited for whatever task you're working on. It takes about 30 seconds or more to load this cell package and the model.

You can look at the Encode function or method from the Sentence BERT and its arguments if you need to evaluate what is useful for your particular task.

Next we're loading a dictionary with famous quotes that we saw earlier. And there are 15 of them. And they are packed into a list of quotes or a list of strings which we will pass to SBERT Encode function to create 15 different 768-dimensional vectors.

We cannot display all of them on screen and scrolling is ineffective so we are displaying just the last 15 dimensions. This is a 15 by 768 dimensional matrix. We're only displaying 15 by 15.

Rows are sentences and columns are dimensions that we do not really understand. We just know that the values that are similar represent similarity of the sentences in that particular dimension. And the mixed values indicate that the sentences are not similar or dissimilar within that particular dimension.

The dimensions will change if you're using a different model. Their are mean will change. And there is active research in understanding these dimensions better.

What we do next is we compute the cosine similarity. Not cosine similarity but the correlation rather. It's a little bit easier. And it's very similar to cosine similarity varying between minus 1 and 1. Similarly, 1 being the maximal similarity between the sentence and itself. And minus 1 being maximal dissimilarity.

This is all packed into a 15 by 15 dimensional matrix which is symmetric. So we have upper triangular and lower triangular being exactly reflected across or around about the diagonal values of 1's.

Higher values in this matrix indicate higher similarity between the two sentences. Like this 0.62 indicates that two sentences are similar and they are having both to do with language and something about vision or eyes.

Next we can query the list of sentences that we have based on their vector representation. And this does not need to have the same-- we do not need to have the same words as we've had previously with TFIDF type of query or document term matrix type of query.

Here the semantic meaning is extracted from the sentence we have, language and sight being the query sentence. And that is represented in the next line with a vector, again, 768 dimensional, which we use to compute every one of these cosine similarities for every one sentence.

Then we package it all as a data frame and order by decreasing cosine similarity so the sentences that are most similar to language and sight are at the top, which they are.

In this video, Professor Melnikov refers to the recent development of SBERT and active research on better understanding the matrix column dimensions it produces. This statement was made in April 2021.

Back to Table of Contents

Code: Practice Embedding Sentences Into Vectors

Previously, Professor Melnikov demonstrated how to use Sentence Bidirectional Encoder Representations from Transformers (SBERT) to encode sentences. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice searching and comparing documents by applying SBERT.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: Compute Similarity of Sentence Embeddings To Find Similar Movies

Now you will practice using SBERT on a data frame of movies and their attributes. Follow along with Professor Melnikov in this video as he uses a pre-trained model to evaluate the cosine similarity of various movies based on their descriptions and attributes. Then, he demonstrates how to display movie results based on a particular query.

Video Transcript

Next we're going to generate movie embeddings from attributes and descriptions in the movie database. And we'll find movie titles from an arbitrary movie description query.

In contrast to previous searches where we use TF-IDF or document term matrix, here we embed raw text without any preprocessing and still produce amazingly relevant search results. Let's see this in code.

We are loading the CSV file with the movies or movie attributes. It's zipped from a GitHub into a dataframe. So here we have this dataframe which is transposed on its side and records are displayed as columns.

The movie *Avatar* has these particular attributes. There is an original title set as an index. There is a title below that is movie title. But also some textual fields like the overview and the tagline right here, "Enter the world of Pandora."

Some other fields are strings but they appear as lists-- sorry, as list of dictionaries. This dictionary is what we call JSON format or JSON structure. It has a key and a value. And we will parse this out. And the reason for parsing is to speed up processing vectors. It does not impact quality very much.

And this is a function that will actually do the parsing. An example here is taking a string which contains all these different structures packaged together and returns either the extracted names of the genres or actual list of genre strings.

There is a bit of regex parsing that goes into this function. But the key element is when we take a structure as a string, pass it into the JSON package or library, and we indicate which key name in particular we want to extract out of that structure.

And in this cell, we are our packaging all of the textual attributes into a single string describing the movie. Notice that we place spaces in between the attributes so that they are not concatenated character to character so that *"Avatar"* word is still separated by a space from the tagline.

The order doesn't really matter. The embedding model will consume this whole text and spit out a vector. Here is the embedding model the Sentence BERT is our favorite. We are loading the Distilled BERT base. It's about 250 megabytes.

Downloading the model takes about 10 seconds. And loading the model is relatively fast. But processing 5,000 movies takes another 10 seconds.

We are displaying this as a dataframe. So we have 768 dimensional vectors for every movie. Here is the first row or first vector of the movie representing the movie *Avatar*.

Now we don't what these values mean or what these dimensions are. But we do know that the movies are similar in some respect if they have similar values, positive or negative, within a particular column.

And now we can finally query that table of vectors or array of vectors by using different query strings. Here we're using the query C. And it could be different capitalization, could be misspelled. We are finding reasonably relevant movies.

So *The Shallows, 20,000 Leagues Under the Sea, The Oceans,* and so on. If we add deep sea, then we'll be finding movies that are relevant to that description.

So here is *The Abyss, The Oceans*. And we can also search for, let's say, moon or Mars. And we'll find movies relevant to that query. And the cosine similarity ranging between negative one and one will indicate the degree of similarity of the returned movie titles or their vectors.

These are ordered in the decreasing order so we can actually see some of the top results. We're picking up the first 10. And they are relevant to the movie query.

And later, the similarity decreases on some of the negative. And actually many of the zero correlation movie results are not alike or dissimilar to the search.
Back to Table of Contents

Code: Practice Computing Similarity of Sentence Embeddings To Find Similar Movies

Previously, Professor Melnikov demonstrated how to use a pre-trained model to evaluate the cosine similarity of a dataset of movies based on their descriptions and how to display the results. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice using SBERT and computing the cosine similarity on a movie dataset for different queries.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: Methods for Clustering

You can use clustering, an unsupervised method for unlabeled documents, to group documents according to their similarities. Watch as Professor Melnikov describes what the clustering process entails and introduces two standard clustering methods: *hierarchical clustering* (HC) and k-*means clustering*.

Video Transcript

Clustering is an unsupervised method for unlabeled documents, where clusters, or groups, of similar documents are identified based on their intracluster similarity and inter-cluster dissimilarity. Generally, the absence of labels makes this task more difficult than the supervised methods we saw earlier, requiring domain expertise and subjective judgment. Even in choosing the number of clusters.

Sometimes, we can manually label small subset of documents to automate model selection and hyperparameter tuning, but numerous clustering metrics also exist. For documents represented as numeric vectors, a Euclidean distance is a popular choice of dissimilarity metric due to its attractive mathematical and computational properties.

There are many clustering algorithms for various situations. But here, we focus on two standard and popular methods, hierarchical clustering, or HC, and k-means clustering. One difference between these techniques is that k-means requires us to specify the number of clusters, k, in advance, while HC requires this after its completion.

Back to Table of Contents

Watch: Hierarchical Clustering of Movies Based on Their Descriptions

You can use hierarchical clustering to group documents based on their vector representations. Although the document vectors often have more dimensions than is possible to visualize, you can rely on distance metrics to determine the similarity between documents.

Libraries/Methods

Standard Library

- `collections.counter()`
- `AgglomerateClustering`
- `pd.DataFrame()`
- `sklearn.cluster, json`

Note: Consider watching these videos in full-screen mode so you can see the Python commands clearly.

Hierarchical Clustering

In this video, Professor Melnikov outlines two algorithms used to build hierarchical clustering trees, or dendrograms: the **agglomerative** approach and the **divisive** approach.

Video Transcript

Hierarchical clustering can be used to group documents based on similarity of their vector representations. The vectors can be drawn from either sparse, TFIDF matrices, or documents or matrices, or dense vector embedding matrices such as bird.

So the documents can be 768 dimensional numeric vectors in the vector space, which we cannot even visualize. However, we can still use distance metrics in this space, such as Euclidean metric or Manhattan metric. And the more similar documents have relatively smaller distance among them.

There are two algorithms to build hierarchical clustering, trees or Denver grams. In agglomerate, or bottom-up approach, we start with every point in its own cluster and merge the closest clusters at each iteration until all clusters are combined into one. In divisive, or top-down approach, we start from all the document vectors in a single cluster. And then break them up recursively until each vector is in its own cluster. These algorithms create a tree, or hierarchy, with branches containing similar documents represented as the bottom leaf nodes.

Hierarchical Clustering Example With Movie Genres

Here, Professor Melnikov uses hierarchical clustering to classify movies from a database into two categories according to their genre. After generating two clusters with `scikit-learn`'s `AgglomerativeClustering` library, he then uses the embeddings demonstrated in previous videos to determine the distance between movies and categorize them. Professor Melnikov classifies Western and Action movies with a “0” and Animation, Family, Comedy, and Fantasy movies with a “1”. He then demonstrates how to plot these findings and identify false positives.

Video Transcript

We will use our movies database to create two sets of movies that we will cluster. And the movies will be drawn based on their genres. So let's look at the genres that we have in the data frame.

There are strings containing-- there appears to be a list of dictionaries but it's actually a string. These are JSON values or JSON structures that we can parse out with our JSON values function which is applied to every value of the genres column.

Once the JSON values function is applied, the end result are all the names of the genres packaged together as a string. Then we can concatenate them all together. And this contains multiple duplicates, of course. And create a statistics or distribution of frequencies across different unique genres.

The movie can have multiple genres. So the movie will contribute-- any given movie will contribute to multiple frequencies here. But we can now combine these genres into something we'll call super genres.

Let's create a dictionary of Boolean list to make it easier. The Boolean list or Boolean vector here has Boolean true or false values, just as many as we have movies. And the true will indicate the positions or the movie roles in our data frame which fall into that specific genre, in this case war.

Now we can create a super mask vector for the animations. And it also is animation, family, comedy, and fantasy. And a Western super mask which contains Western and actions.

So we are creating a masking vector which is animation, not, the Western, and the second one is Western, not animation. And then combining into-- combining those into one single common super masking vector.

In total we have 30-35 movies in each of the groups that we've created, total 65. So this is presentable. We can plot this nicely. They will not over plot and we can still understand the meaning of these movies.

We know what these movies are. Some of them are animations. Some of them are Western. And that makes sense. We'll use agglomerative clustering library in scikit-learn package.

Number of clusters is two. And that's because we know we have synthetically created two groups of movies. So we're hoping that agglomerative clustering can find this.

And then we're feeding our embeddings that we created earlier with this agglomerative clustering. Meaning that it learns the distances and clusters from these embeddings.

There are default parameters that we'll investigate later. For now, we are presenting these learned clusters for each movie. So zero should correspond, for the most part, with few exceptions or false positives to actions and Westerns. And cluster 1 are animation, family, comedy, and fantasy.

We cannot plot 768-dimensional vectors but we can compress that into two-dimensional space using PCA, Principal Component Analysis, which uses Singular Value Decomposition or SVD underneath.

We do need to specify how many components we want to draw and the most important components are retrieved, which will be our dimensions. They are X and Y or PC1 and Principal Component 2.

Next we'll create an array of colors. There are only two colors that alternate depending on cluster number, zero or one, that is assigned to a movie. Because these values are RGB or Red, Green, and Blue shades that will be used as colors in the Plotly package.

The Plotly package will display all these different points and the colors of the clusters that have been assigned to these points. We can see that the blues are primarily the Western action movies. The Reds are primarily animations, family, fantasy.

But there are some false positives where Delgaf appears to be in the cluster of reds but is mislabeled or it's been assigned to the wrong cluster.

Back to Table of Contents

Watch: Practice Hierarchical Clustering of Movies Based on Their Descriptions

Previously, Professor Melnikov demonstrated two algorithms for building hierarchical clustering trees and then used agglomerative clustering to classify movies from a database into two categories using embeddings and distance calculations. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice clustering movie vectors using agglomerative clustering and analyzing your results.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: The Anatomy of a Dendrogram

Grouping points with a hierarchical clustering algorithm can produce a **dendrogram**, or tree diagram, as a result. You may already be familiar with decision trees, which model classification possibilities for labeled data in supervised learning. Dendrograms are similar but pertain to unlabeled data in unsupervised learning.

Libraries/Methods

Standard Library

`scipy.cluster.hierarchy`

`Dendogram()`

`AgglomerativeClustering()`

`PlotDendrogram()`

In this video, Professor Melnikov creates a dendrogram using the `PlotDendrogram()` function and then describes its different parts using movie genres from the previous hierarchical clustering example. He also demonstrates how to use dendrograms to evaluate clustering quality and visualize data to determine an appropriate number of clusters.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

A dendrogram is a tree which results from grouping points in a hierarchical clustering algorithm. It starts with a root at the top of the tree and branches down one split at a time. The bottom nodes contain single observations are leaf nodes. The remaining nodes are called interior.

The length of a path between two clusters or nodes is the sum of its edges, and is proportional to the distance between these clusters in the vector space. So longer paths in the dendrogram indicate longer distances. We can use dendrogram to evaluate the quality of clustering and the appropriate number of clusters. Let's see a dendrogram built on a movie database in code.

We have this function called plot dendrogram, which takes the model hierarchical clustering and movie titles that will be used as leaf labels. Then we're creating agglomerative clustering object once again. Except here, we want the tree to grow all the way down. We don't want it to be truncated at any number of clusters. So we're specifying number of clusters as none.

Finally, we're plotting the dendrogram. And it takes a couple of seconds to appear. Here we have a dendrogram which is a tree, from the top root node, which is not indicated here, to the bottom leaf nodes, which are movies.

Overall, the structure seems reasonable. We do see reasonable clusters, green and red. And if we look closely, the green appear to be mostly animation and the reds appear to be mostly westerns and actions, but not necessarily. There are some False positives that are possible.

This visualization allows us to determine how many clusters we might want or need. So here, it seems that the blue line, which indicates the distance, the Euclidean distance between green and red, larger clusters is the longest. And it makes sense to put a threshold and cut this tree into two branches, or two individual trees, at the blue line.

The more optimal threshold would be somewhere in the middle because the nodes and the structure of the tree will change a little bit if you keep adding or removing movies from it. In a multidimensional space, these points may be located maybe having different clustering relationships as the algorithm builds the tree. So if you stay away further from values like 23 or value 40 and stay in the middle and threshold it was a horizontal line, then you will have more reasonable, more stable two clusters.

Now you can lower the threshold to about 20. And then you will create four clusters. So somehow, this movies in the cluster, perhaps green one, green two are somewhat distinct. They are vectors embeddings indicate some semantic differences. And same goes for the cluster, this red one and this red two at this threshold.

Now, it is up to you to decide how many clusters you want to create. Typically, we would look at the longest path and cut that off. But some domain knowledge is useful. And we can see that movies like *Shrek* create their own subcluster. *Shrek* and *Shrek 2* are right next to each other. And the length of the green edge between, or path between them, indicates their proximity in the vector space.

And if you have a new movie that you want to label or assign a genre to, what you can do is you can push that movie through this dendrogram. And as it approaches a particular node, you will decide which cluster is closest to, this cluster or this cluster, and put that movie into that cluster.

And then continue on this making those decisions until it lands in some smaller cluster. And then you can use that clusters label, smaller cluster or aggregated cluster label, to determine the genre of that movie. That's how we can use this dendrogram.

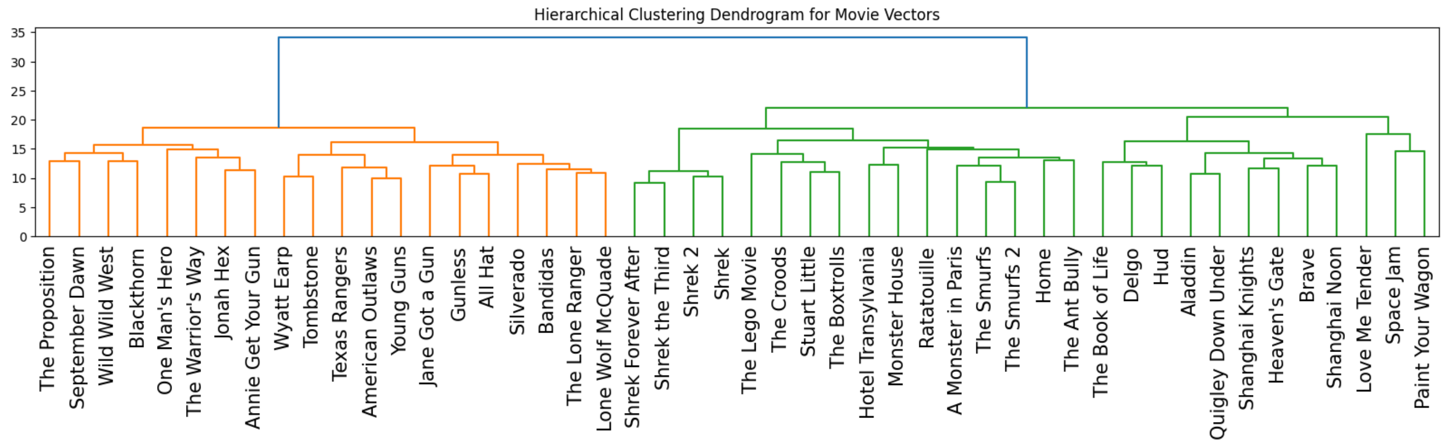
Back to Table of Contents

Quiz: Interpreting a Dendrogram

In this module, Professor Melnikov has introduced a **dendrogram**. Now it is time to test your understanding of the concept. Select your responses, keeping in mind that a question may have more than one correct answer. Submit your quiz when you have finished. You may take this quiz as many times as you would like. Your best score will be included in your course grade.

Refer to the following movie dendrogram for some of the questions, a dendrogram of movie titles generated with Ward's linkage. It has seemingly two large clusters identified by the long blue edges coming from the top.

1. 1st split (into orange and green) occurs at about a vertical threshold value of 34.
2. 2nd split (of greens branches) occurs at about a vertical threshold value of 22.5.
3. 3rd split (of green branches) occurs at about a vertical threshold value of 20.5.
4. 4th split (of green branches) occurs at about a vertical threshold value of 19.
5. Shrek movie merges with Shrek 2 movie, which forms a subcluster, which then merges with another subcluster containing Shrek Forever After and Shrek the Third.



This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve the maximum score.

Please complete this activity in the course.

Back to Table of Contents

Watch: Build Dendrograms Using Different Linkages

Since a cluster is not a single point but spread out over space, there is no natural inter-cluster distance. In order to find the distance between clusters, you can use different inter-cluster metrics, or **linkages**, to find the distance between two points, A and B, in different clusters. Each linkage method has a different approach for selecting points A and B.

In this video, Professor Melnikov describes several linkage methods and discusses some pros and cons of each. He then shows the resulting dendrograms in code, using the movie data to model different linkage methods.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

We can compute Euclidean distance between two points in a vector space. But there is no natural distance metric for clusters because clusters are spread out in space. They are not single points. There are several inter-cluster metrics called linkages, however.

Say we need to identify point A in one cluster and point B in another cluster so that we can compute the distance between points A and B as the inter-cluster distance, and we have few options. The minimum, or single, linkage finds A and B, which are closest to each other. Remember that A and B should still remain within their clusters. So we're looking for points that are closest between the two clusters, but within their clusters.

As a side effect, this tends to combine, or chain, series of close intermediate points, which makes thresholding of distinct clusters difficult. The maximum or complete linkage finds A and B, which are maximally distant from each other. This is very opposite to the previous linkage metric, or function.

This tends to produce many small clusters, which have observations that are more similar to observations in other clusters. Also, not very ideal for stable clustering method. There is also centroid linkage. And it computes A and B as the cluster centers. Recall that a centroid is a set of vectors of a set of vectors is just a mean or average vector. It basically averages the vectors element wise.

This linkage can result in dendrogram inversions, where the edges of the dendrogram would cross each other. And this complicates the interpretation and thresholding. The group average is another way to measure the distances between clustering. Uses the average distance among all paired combinations of points in two clusters. It is a compromise between single and complete linkages.

However, unlike those group average, clustering is sensitive to monotone transformations of the distances. So if you need to apply a log or logarithmic transformation or square root transformation to the distances, you may develop completely different set of clusters.

Scikit learn uses Ward's linkage as its default. And it works fairly well in general in merging clusters it tries to minimize the intracluster sum of squared distances of points to their centroids. There are many other linkages as well.

Ultimately by trying and learning about this linkages, you will develop an intuition for their use. Also in addition to Euclidean distance, we could use other distance metrics such as Manhattan, or L1 distance, which is less sensitive to outliers because it doesn't square the values. So if you have that, you have outliers as a problem, you might switch to L1 distance. Let's see some of these examples in code.

The first example uses a single linkage. And if we create a dendrogram based on it, we will see how it links, or chains, together a single cluster and just keeps adding points or keeps adding movies to it. The *Shrek* movies are still clustered together, but there is no single threshold, horizontal threshold, that would create two distinct clusters.

The next one is complete linkage. So we're, again, creating the dendrogram with complete linkage. But the edges, or the path, are too short or too shallow. So there's very little space or vertical margin to put the horizontal threshold in and reliably cut off to two clusters. And there's also worries that this clustering will slightly change and the thresholding will significantly cut off or recombine different clusters. The slight changes could be from adding more movies or removing movies from this set of points.

There's another clustering called average link. And it has a very similar problem where the edges are too short. And finally, the ones that is default and probably behaves the best in any default scenario. And that's the word linkage, which presents a very nice long blue path between two clusters. And if we find the center point on the vertical line to threshold is two clusters, that they should be reliably reproduced with other examples of the same movie types.

Back to Table of Contents

Code: Practice Building Dendrograms Using Different Linkages

Previously, Professor Melnikov demonstrated how to create a dendrogram using the results of hierarchical clustering to evaluate the quality of the clustering and how to use different linkage methods to find the distance between clusters. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice clustering movie vectors and how to determine the better linkage method by viewing a dendrogram.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

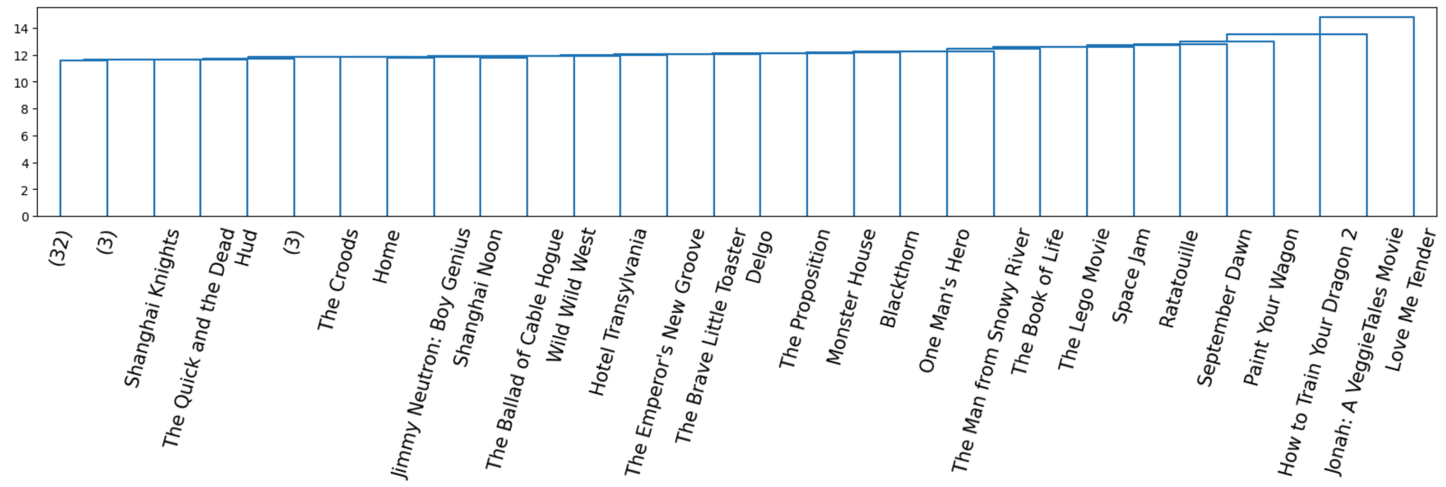
Quiz: Identify Linkages in Dendrograms

In this module, Professor Melnikov has introduced **linkages**. Now it is time to test your understanding of the concept. Select your responses, keeping in mind that a question may have more than one correct answer. Submit your quiz when you have finished. You may take this quiz as many times as you would like. Your best score will be included in your course grade.

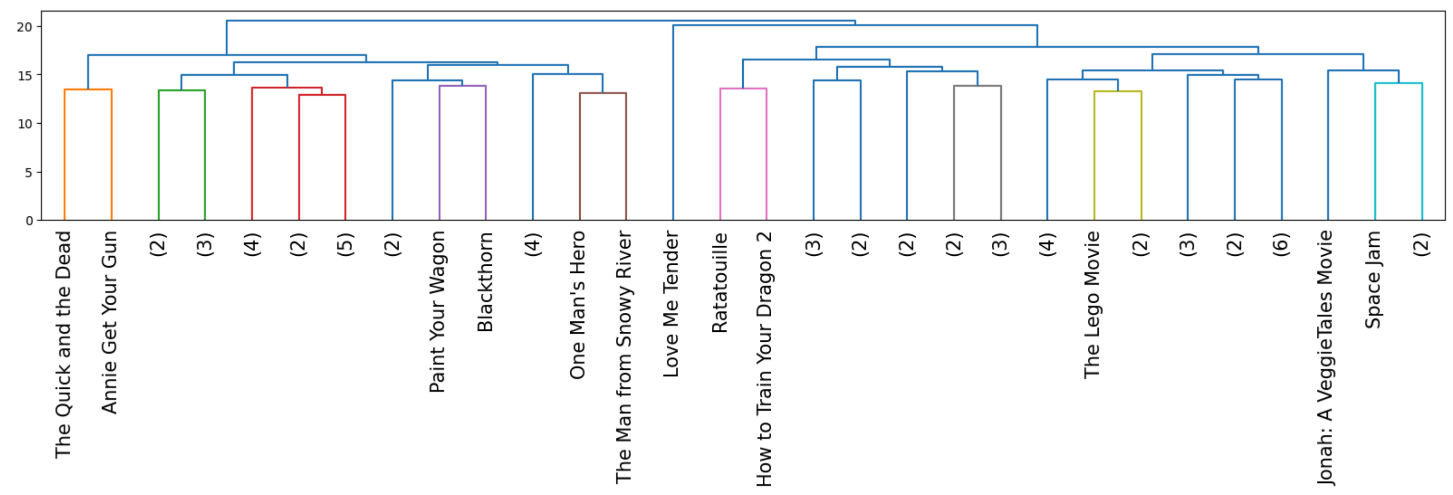
For the questions below consider the following:

1. Suppose we need to identify point A in one cluster and point B in another cluster, so that we compute the distance between A and B as the inter-cluster distance.
2. The numbered dendrograms below. The numbers in parentheses show the leaf count under the given brunch. E.g. In Dendrogram 1, (32) indicates that there are 32 more movies under that brunch.

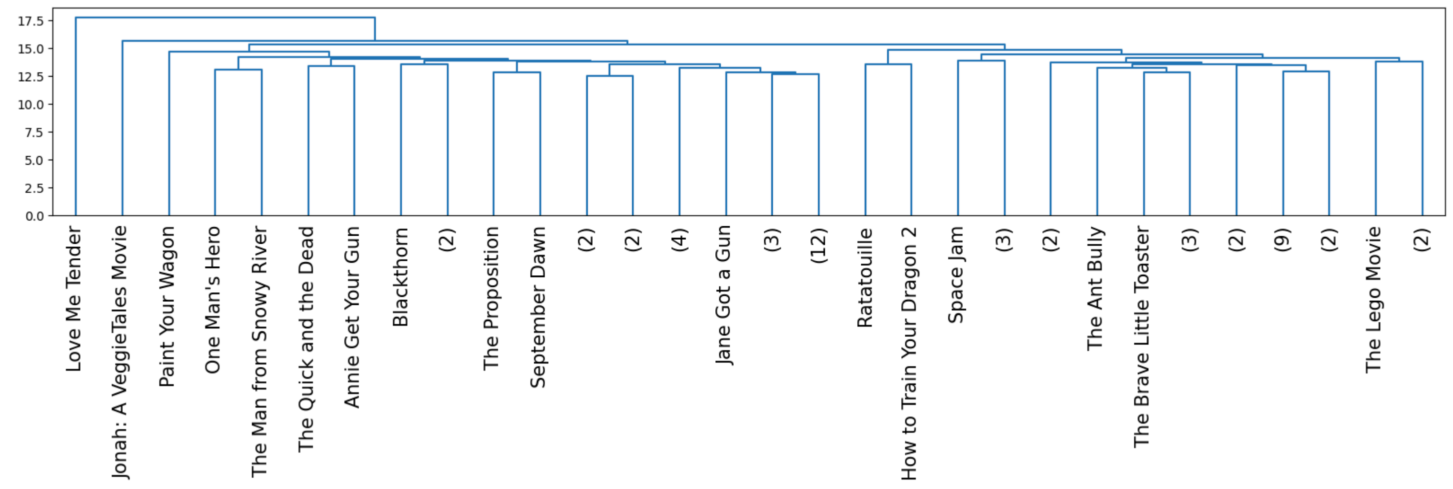
Dendrogram 1 (with a single large cluster). The edges to leaf nodes are short. So, sliding the threshold to zero gradually increases one cluster and shrinks another.



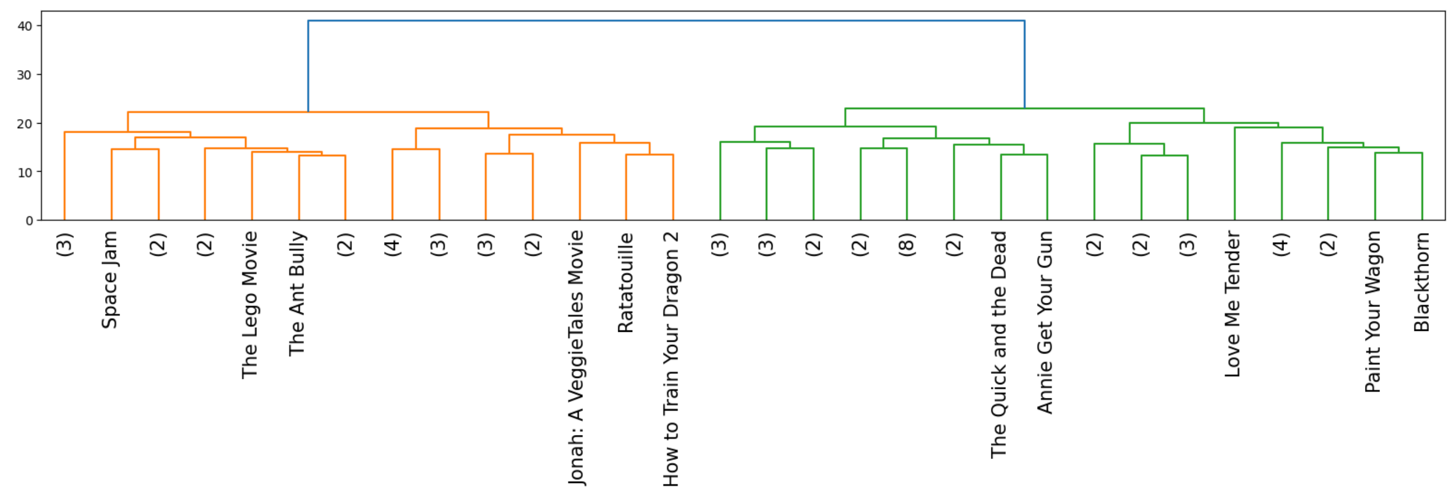
Dendrogram 2 (with many small clusters). It has many short top edges. So that thresholding is likely to produce many dispersed small clusters.



Dendrogram 3 (with possible inversions). Also has many short top edges, but doesn't tend to produce many dispersed clusters. A distinct feature is a possibility of inversions.



Dendrogram 4 (with long top edges coming from the root). Long top edges so that most thresholds at the top will likely produce stable clusters, which do not change their memberships.



This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve the maximum score.

Please complete this activity in the course.

Back to Table of Contents

Watch: Measure Clustering Performance

There are two main types of assessments for measuring cluster performance: qualitative assessments and quantitative assessments. However, since qualitative assessments are usually subjective and difficult to perform on a large scale, in this course you will primarily focus on measuring your cluster performance using quantitative assessments.

Here, Professor Melnikov demonstrates how to compute several quantitative metrics in code. In this example, he uses agglomerative clustering to group a set of movie vectors into two clusters. He then identifies the accuracy score, Rand index or adjusted Rand index (ARI), and silhouette score as three metrics to describe the clustering performance.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Cluster evaluation is non-trivial. Even if we have two class labels, it is unlikely that they would be aligned with the other assigned cluster labels.

For example, it is difficult to reconcile five movie genres with, say, seven other discovered movie clusters labeled zero through six. The algorithm just doesn't know whether zero represents an action genre or some combination of family and comedy. This challenge is also an opportunity for several clustering metrics to emerge.

Quantitative metrics can be either extrinsic where we evaluate how well clustering improves some downstream task like classification. Or intrinsic, where we quantify intracluster variability and scale it by intracluster distances.

Qualitative assessment is basically looking at a plot with marked or colored clusters to subjectively decide on whether they are well represented or well grouped.

But doing so for hundreds of cluster variants is time consuming and to be honest, tiring. Let's see some of the quantitative metrics in code.

We're creating an object agglomerative clustering here with two clusters and feeding it on the set of movie vectors. Two clusters seems to work very well because we actually manually allocated or picked out movies that should cluster well into two clusters. We're using the default linkage word with the Euclidean distance.

In the next cell, we are manually aligning the cluster labels with the class labels. It is mostly aligned but we have some misalignments somewhere.

Because these clusters are balanced having approximately the same number of points and the classes are balanced, we can use, meaningfully, an accuracy score.

An accuracy score would tell us the proportion of matched labels and it's 95.4% of matched labels, which is meaningful considering that the blue points and the pink points in our scatterplot of movie representing vectors into dimensions were fairly well clustered but with some overlap.

If we don't have the labels, we can use something called Rand index. It is developed by statistician William Rand. And it counts all pairs of points assigned to the correct clusters and those assigned to the incorrect clusters.

It relies on true labels which are not always accessible. And in that case, human experts can look at the samples of paired points. And then there is an Adjusted Rand Index and it's ARI.

And basically, re-scales Rand index to be an interval of zero-one where zero indicates no pattern and one indicates perfect clustering. The adjusted Rand index here is 0.82 which makes sense considering that the clusters are fairly well separated.

The silhouette score relies on intercluster distances or variance and intracluster distances. A represents, in the formula, The mean intracluster distance. And B represents the mean distance to the nearest cluster.

If we subtract A from B and divide it by the maximum of the two, we get 0.1. This is fairly low considering that the metric stays in minus one to one interval. But remember that the clusters do overlap a little bit and some of the blue points are in the wrong cluster.

However, it alleviates us from aligning manually all these different cluster labels and class labels. And we can use any one of these cluster metrics if we have them accessible in automatically picking the number of clusters.

So we have the score clusters function which takes K as the number of clusters. It re-computes the agglomerative clustering with that K, refits the object on this movie vector set, and computes two different metrics.

And if we plug this all, we see that the higher value, which is at K equals 2, tends to favor the two clusters that are reasonably well separated.

Code: Practice Measuring Clustering Performance

Previously, Professor Melnikov demonstrated how to compute a variety of quantitative metrics to measure cluster performance from agglomerative clustering and showed how results vary with a different number of clusters. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice writing a function to compute accuracy scores to measure cluster performance and determine the ideal number of clusters.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Assignment: Course Project, Part Two — Performing Hierarchical Clustering on Sentence Embeddings To Group Similar Texts

In Part Two of the course project, you will be encoding movie descriptions into numeric vectors in order to cluster them with various sets of parameters and determining the best set of parameters for hierarchical clustering using the silhouette score. The tasks you will perform are based on videos and coding activities in the current module, but may also rely on your preparation in Python and basic math.

Use the tools from this module to help you complete this part of the course project. Additionally, you may consult your course facilitator and peers by posting in the project forum that is linked in the sidebar.

Completion of this project is a course requirement.

Resources

Use these resources to help you as you complete the course project:

- **Project Forum**
- **Jupyter Notebook Guide**

Instructions

Read the general project instructions within this accordion carefully, then follow the instructions specific to this part of the course project in the Jupyter Notebook below.

Please complete this activity in the course.

This exercise is **graded**, and may take ~3 hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Perform Hierarchical Clustering on Sentence Embeddings To Group Similar Texts

In this module, you practiced converting documents to numeric vectors using SBERT, which is an important first step before categorizing documents into categories with clustering. You then applied hierarchical clustering to a dataset of movies, categorizing movies into two clusters according to genre. Finally, you evaluated the performance of your clustering using dendrograms and three quantitative metrics: the accuracy score, the Rand index or adjusted Rand index (ARI), and the silhouette score.

At the beginning of this module, you were introduced to two main clustering approaches. Now that you've practiced using hierarchical clustering to categorize your documents, you will move on to another approach: *k*-means clustering.

Back to Table of Contents

Module Introduction: Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts

So far, you have explored hierarchical clustering and practiced embedding sentences as vectors using SBERT. In this module, you will continue to use SBERT to represent texts as vectors. However, now you will be introduced to *k*-means clustering, which you will practice throughout the module. After using *k*-means clustering to categorize movies, you will evaluate the quality of your clusters using silhouette scores.

You will also encounter another type of problem in this module: identifying an item within a set that is representative of the set as a whole.

Back to Table of Contents

Watch: Calculate Centroids and Medoids To Find a Representative Data Point

A centroid is the mean vector of a set of vectors and is used to represent a set of vectors. Importantly, a centroid is not the same as any of the vectors in the set it represents. When you apply this concept to a corpus and you want to find a representative document in the corpus, finding a centroid will not work as it is the mean and does not correspond to one of the documents. A medoid, the closest existing vector to the centroid, allows you to find a representative document, which can be used to help determine what labels should be applied to the cluster. Watch as Professor Melnikov defines these terms and plots examples that compare the centroid and medoid for different sets of vectors.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

A centroid is the average vector of a set of vectors. For the scalar values 1, 2, and 3, the centroid is just the average of these numbers, or 1 plus 2 plus 3 divided by 3 equals 2. In a vector space, a centroid is an element-wise sum of vectors divided by the number of these vectors. It's just a main vector.

A centroid is rarely a point of the underlying set of points, which may not make sense if you're looking for a representative document in a corpus. That's what a medoid is. A medoid is the central representative point of the cluster. And it must be one of the cluster points. Computing medoids is slower since we first compute the centroid and then we find the existing point closest to the centroid. Let's see some examples in code.

We are defining four vectors-- a, b, c, d-- in a two-dimensional space, and computing the centroid as the mean of this vectors. We can equivalently just sum all the vectors and divide it by their count. The get medoid function will take matrix, where the rows are vectors, and it will compute the average vector or the centroid for those rows, and then it finds the vector closest to that centroid.

We are not using the Euclidean distance. We're using the squared Euclidean distance, which will give exactly the same result. The square root here does not change the answer. Notice that the centroid is not any of the points, but the medoid is. It's the point D.

And we can plot this on a two-dimensional surface, where the blue points are the original points a, b, c, d. The red one is the centroid. And the crossed point is the medoid, which is one of the points in the set. We can create a three-dimensional picture of the points in space, which is a bit difficult to view. Not my favorite.

The shading of the blue probably represents the depth here. But typically, the depth is difficult to tell. The orange point is the centroid. And these points are generated from a Gaussian distribution with zero mean and one variance.

The next sample we draw uses make blobs function in scikit learn. And it generated 150 samples, or observations-- points-- in two-dimensional space. So there are two features. And from three different clusters, each one is from the Gaussian distribution with the standard deviation .0.6.

And this is the data set or data frame with all the different points, where y indicates which cluster x1 and x2 belong to. Now these are not the clusters we find. These are clusters that we simulate. So oftentimes, we use make blobs to feed to a particular algorithm and see how well that algorithm finds or determines, or predicts the cluster labels and compare them to the ones that were simulated.

Here, we're just using it to compute the mean for every cluster. And this is done by grouping observations by the y column and computing the mean on each of the groups. And we are computing medoids in a similar fashion. We're grouping by the y column and computing get medoid function on that particular group to find the medoids.

These medoids are actually points of the original set. That's a very important point of this definition. Then we're computing the global mean, which is the mean of all the points from all three clusters. And the global medoid, which is, likewise, the central point, which is point of the set. But it's the central to all three clusters.

There's a function adjust lightness which will take a color in as a string name or RGB-- red, green, and blue, or HMS, or other representations-- and it will make it lighter, brighter. We'll use it to identify the central points. And it returns the RGB tuple with the numbers representing the red, green, and blue.

Finally, we will plot all the points with their respective colors. And we'll plot the larger points as the centroid and medoid. The medoid is the ones that has the point lying over it. So this red point is the medoid. It has a point-- some point lying on top of that. Notice that if we're computing medoids from incorrect set of clusters. So here we're computing this red point's centroid and medoids from combined clusters.

Then they are not they may not be in the clusters themselves at all. So they are not very representative. But the cluster centers that are laying inside of their respective clusters are much more representative of their clusters.

Back to Table of Contents

Watch: Practice Calculating Centroids and Medoids To Find a Representative Data Point

Previously, Professor Melnikov demonstrated how to compute the centroid and medoid for a set of vectors and he used a plot to compare these two methods to represent the data. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice calculating the centroids and medoids for different groups of points.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: Compute the Medoid To Find a Representative Movie

You can use the medoid to find a representative movie from the movie database for a specific genre. In this video, Professor Melnikov demonstrates this technique for action movies in code.

Follow along as Professor Melnikov loads the movie database, uses JSON to parse and clean up fields, and filters out only the action movies. He then selects a single representative movie by identifying the medoid for a matrix containing vector representations of all action movies in the database. To visualize the results, he uses Principal Component Analysis (PCA) to compress the vectors from 768 dimensions to two-dimensional vectors and then plots the results.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Libraries/Methods

Standard Library

`SentenceTransformer()`
`JSON`, `SBERT.encode()`
`pd.dataframe`, `np.mean()`
`np.argmin()`, `PCA()`

Plotly Library

`scatter()`, `layout()`
`figure()`

Video Transcript

The common task in NLP is finding representative documents. Let's look at a movie database example where we find a representative movie as a medoid for a specific genre cluster. In the first, cell we are loading the sentence transformer or SBERT package and model, or package and model is DistilBERT base. It's about 250 megabytes and it takes roughly 20 seconds. 10 for downloading it and 10 for loading the model.

And in the next step, we're downloading the movie database from a GitHub location, and parsing it accordingly. Now, there is this JSON values function that will take a list of JSON or it's actually a string of JSON structures. And extract the values given the specified key. We will use that to clean up some of the JSON fields like keywords, the production countries, the genres, and we also save the cleaned up genres in a separate column.

This looks like-- so this is the cleaned up database where we have a title of the movie as an index. We have a description that describes the movie. And we have genres. And we're also filtering the 5,000 movies to just about 1,000 of action movies. Within these movies we want to find the most representative medoid. In the next cell, we are parsing this list in particular the description column to the SBERT encode method.

And it will create a matrix of 768 dimensional horizontal or row vectors for each one of the movies. So we have 1154 movies by 768 dimensions. We can now use this vector representations for the movies to compute the medoid. And for the medoid, we're using very similar function as before, except it returns an index not the vector. And the index of the medoid, we can use that index-- yeah saved here as NIX to locate the movie that is most representative for this action set of movies. And this movie is named *The Hunting Party.*

Now, we would like to visualize this cluster of movies and see where *The Hunting Party* is located in this cluster visually, but we cannot do this in the original 768 dimensional space. So we use PCA-- Principal Component Analysis, which uses singular value decomposition or SVD to compress 768 dimensional vectors to just two dimensional vectors. Now, every vector is represented as an approximate point in the two dimensional space. So points with axis names x and y.

Now, we can plot this with the plotly object. And plotly allows us to have labels over the points. So we can actually visualize the movie titles. And we are adding the medoid to the existing scatter plot. So here is our medoid in orange, and all the other ones are the action movies. We see them scattered all around. And this movie is reasonably in the center, *The Hunting Party.*

We can actually zoom in and see some of the movies around it. So *A Lonely Place to Die,* and I'm not sure what year this is. *The Hunting Party* is fairly old. I think it's 1970s. *Dead or Alive* is right next to it. So there's are similar movies, or movies having this similar description in the movie database. And that's how we find the representative document for this particular cluster movies.
Back to Table of Contents

Code: Practice Computing the Medoid To Find a Representative Movie

Previously, Professor Melnikov demonstrated how to select a single representative movie from a dataset by computing the medoid for a matrix that contained the vector representations of all of the movies in that dataset. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice using SBERT and calculating the medoid for different datasets of movies.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Watch: Clustering Movies With K-Means and Evaluating Performance

A popular clustering algorithm is ***k*-means clustering**, where *k* is a specified number of clusters. The algorithm will randomly select *k* points from the sample and all of the other points are assigned to one of the initially selected *k* points, resulting in all of the points being assigned to one of the clusters. The final result will be the centroids that are either repeated most frequently or that yielded the best clustering metric. You can also test different numbers of clusters by trying multiple values of *k* and computing and comparing the silhouette scores to find the best number of clusters.

Libraries/Methods

Standard Library

`pd.dataframe()`

sklearn library

`kmeans()`, `silhouette_score()`

`random.choice()`

`pd.DataFrame()`

plotly Library

`scatter()`, `layout()`

`figure()`

Follow along as Professor Melnikov demonstrates how to use this algorithm for the subset of action movies from the database in code. He first shows an example with a pre-selected number of clusters and then shows how to determine the best number of clusters for the dataset.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

K-Means is another popular cluster in algorithm. First, we must specify k as the number of clusters to find, then the algorithm initializes with k random points in the original feature space, which eventually converts to k centroids representing k respective clusters. At each iteration, the algorithm uniquely assigns all points to their closest centroids, and then re-estimates k centroids from the groups of assigned points.

The algorithm is guaranteed to converge, but not to the unique global set of central rates. This is a problem. To overcome the algorithm sensitivity to initialization, we may initialize it many times, say 100, and then pick the centroids that are repeated most frequently, or that yielded the best clustering metric. This random start initialization would be repeated for each k if we also wanted to determine how many clusters to use.

Let's see k-means example in code. We're loading k-means and silhouette score, and create a k-means object, and then feeding it on the set of movie vectors. The parameters are three clusters. And then n jobs equals minus 1 indicates that we want to use all CPUs available to us. The random state is 0 is fixed in initialization, at whatever random set of three initial centroids that we start with.

Maximum iteration is a number of iterations that we should not exceed. And number of initialization is 30, meaning that we will initialize this 30 times and then pick the best of the 30 centroids that we find. Here are all the specifications.

We can see how many iterations have been executed, and that's 16, not 1,000. So it converges reasonably fast for a medium size of set of points. We can also extract the labels and assign those labels to all the observations, all the documents or movies that we had originally. Here, we compute the frequency of observations within each found cluster. So the largest cluster is number one, the number in here doesn't-- or name doesn't really matter. We can call it 1, 2, and 0, or 0, 1, 2. But the largest one has 430 movies. Next largest has 367, and so on.

Then we can show the centroids. Remember that k-means finds the centroids, not the medoids. So we have three centroids, one for each cluster. And each centroid is a vector from 768 dimensional space.

The silhouette score will tell us how well this clusters-- three clusters-- are separated, and how dense they are. It's 0.035 metric, which is greater than 0 but it's not quite close to 1. So it's fairly poor clustering that we have here.

Now let's see what this cluster looks like. We'll need colors to label, or to identify, points or movies from different clusters. So we create RGB set of-- or vector of RGB colors-- as strings, and assign those to the plot [INAUDIBLE] objects. So here are the three clusters that we have identified. And to be honest, I probably say that 2 is more appropriate than 3. Green by itself and red and blue are probably better be combined together.

So if we don't know what the key would be, then we'll try different cases we will do in a second. So here we have different movies in green-- drama, and history, science fiction, all of this, or action, so action and something else-- action, thriller. So I see quite a few thrillers, horrors. So maybe this is a thriller set. And red ones are also thrillers. But there are some comedies.

So perhaps a more-- the main knowledge would be helpful here to indicate what these clusters are. But it's not a very well-clustered set of clusters either.

So here we are trying a number of clusters from 2 to 10. And for each one, we would fit 20 k-means with 20 initializations. And we would compute the silhouette score for each one of these k. So the silhouette score suggests that 2 is the better number of clusters. But it's still fairly low-- away from 1. And this is by design. We took all the action movies. But there or action movies-- and they are other genres as well. So it's not-- it will not be easy to differentiate these movies without a better or deeper domain expertise.
Back to Table of Contents

Code: Practice Clustering Movies With K-Means and Evaluating Performance

Previously, Professor Melnikov demonstrated how to select a single representative movie from a dataset by computing the medoid for a matrix that contained the vector representations of all of the movies in that dataset. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice running k-means models with different sets of parameters, computing the silhouette scores, and determining which parameters yield the best results based on the silhouette score.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

Back to Table of Contents

Hierarchical and *K*-Means Clustering

In this course, you have practiced two approaches for clustering movies into genres: hierarchical clustering and *k*-means clustering. Prior to clustering, you have preprocessed your data by embedding sentences into vectors with SBERT. After clustering, you have modeled your findings visually through plotting.

This tool demonstrates the clustering process for both algorithms, from preprocessing to plotting. Use it as a guide whenever grouping data with hierarchical clustering or *k*-means clustering.

Please complete this activity in the course.

Back to Table of Contents

Assignment: Course Project, Part Three — Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts

In Part Three of the course project, you will be encoding movie descriptions into numeric vectors in order to cluster them with various sets of parameters and determining the best set of parameters for k-means clustering using the silhouette score. The tasks you will perform are based on videos and coding activities in the current module, but may also rely on your preparation in Python and basic math.

Resources

Use these resources to help you as you complete the course project:

- **Project Forum**
- **Jupyter Notebook Guide**

Use the tools from this module to help you complete this part of the course project. Additionally, you may consult your course facilitator and peers by posting in the project forum that is linked in the sidebar.

Completion of this project is a course requirement.

Instructions

Read the general project instructions within this accordion carefully, then follow the instructions specific to this part of the course project in the Jupyter Notebook below.

Please complete this activity in the course.

This exercise is **graded**, and takes ~3 hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Perform K-Means Clustering on Sentence Embeddings To Group Similar Texts

In this module, you continued exploring clustering approaches, but also examined a new problem: how to identify a document that is representative of an entire corpus. First, you gained a familiarity with centroids and medoids — two vectors used to identify the average item in a set. You then used these vectors on a data set of movies, to practice identifying a representative action movie.

Finally, you examined another approach for clustering documents: *k*-means clustering. Using your familiarity with SBERT from the previous module, you created sentence embeddings to categorized with this new clustering technique, and then used silhouette scores to evaluate the quality of your clustering.

Back to Table of Contents

Thank You and Farewell

Oleg Melnikov

Visiting Lecturer

Computing and Information Science
Cornell University

Congratulations on completing *Clustering Documents With Unsupervised Machine Learning*.

I hope that you now feel more equipped with the tools needed to identify clusters of similar documents as well as find representative documents in a corpus. Although each concept in this course may appear challenging at first and trivial after learning and practice, overall you are building a valuable expertise in NLP, ML, Python and other disciplines. This investment in yourself will pay dividends for many decades to come. It will open conversations and opportunities into the most miraculous ventures. Keep on learning!

From all of us at Cornell University and eCornell, thank you for participating in this course.

Sincerely,

Oleg Melnikov