CIS572: Transforming Text Into Numeric Vectors

Table of Contents

- Assignment: Course Project, Part One — Creating Sparse Document Vectors
- Module Wrap-up: Create Sparse Document Vectors From Counters

Module Introduction: Create Dense Document Vectors From Pre-Trained Models

- Watch: Dense Matrices and Word2Vec
- Code: Practice With Dense Matrices and Word2Vec
- Watch: Word2Vec Vectors
- Code: Practice With Word2Vec Vectors
- Watch: Arithmetic With Word Vectors
- Code: Work With Word Vector Arithmetic
- Watch: How Word2Vec Is Trained
- Quiz: Model Architecture Basics
- Watch: The Long Tail Problem for Out-of-Vocabulary Words
- Code: Address the Long Tail Problem
- Watch: Generating Subwords With FastText
- Code: Generate Subwords in FastText
- Assignment: Course Project, Part Two — Creating Dense Document Vectors
- Module Wrap-up: Create Dense Document Vectors From Pre-Trained Models

Module Introduction: Measure Similarity Between Document Vectors

- Watch: Similarity
- Watch: Similarity and Distance Metrics
- Watch: Examples of Similarity Metrics
- Tool: Similarity Metrics
- Code: Generate Similarity Metrics
- Watch: Finding Similar Documents With TF-IDF
- Code: Find Similar Documents With TF-IDF

- Watch: FastText Model Formats and Transfer Learning
- Watch: Building Sentence Vectors With FastText
- Code: Build Sentence Vectors With FastText
- Watch: Finding Similar Documents With FastText
- Code: Find Similar Documents With FastText
- Assignment: Course Project, Part Three — Measuring Similarity Between Vectors
- Module Wrap-up: Measure Similarity Between Vectors

Welcome to Your Course

Video Transcript

Previously, we learned operations on text, including tokenization, preprocessing, and counting. A much richer world of operations and representations of text is accessible through mathematical structures such as vectors and matrices. In this course, we develop techniques that allow us to abstract any textual content as a multidimensional point or a vector in space. As an analogy, a vector space can be thought of as our physical world. Then the documents can correspond to people, with their unique personalities and attributes. A simple idea is that two people close to each other are likely to share similar characteristics and even mindsets. They can be both members of the same family, school, employer, soccer team, or even flight to New York. It's amazing how powerful a simple distance measure can be and how much it can reveal about people around us and similar for the documents.

Course Description

In natural language processing (NLP), you often need to compare documents to programmatically find the most relevant document or to group them into categories. Unlike numbers, however, documents cannot be unequivocally compared on a real line. Documents lack a *natural distance* or *similarity* measure that allows you to determine the proximity between documents A, B and C. You therefore need to define a suitable distance for a corpus to accomplish a particular language task and be able to determine whether B or C is "closer" to A and by how much. Once the distance or similarity metric is defined, you can order documents B, C, and a million others by their proximity to A so that the most relevant appear first. This is important in selecting, for example, the top web search results, news posts queries, and a sentence translation candidate, as well as many other use cases.

If you compare a small pool of short strings, such as letters, words, or phrases, a binary comparison can be sufficient. This is when you determine whether two strings (or any objects) are identical (resulting in value 1) or not (yielding a value 0). For example, you

might look for all users with the query name "Oleg Melnikov" and are not interested in any other variant in the pool of names.

As the range and complexity of string objects grow, this metric becomes less effective. For example, in a pool of a billion scientific research articles, it is highly unlikely that any two of them are identical, so a binary comparison would result in 0 for any two articles. Yet many articles would use the same words, phrases, topics, methods, terms, and ideas, so you need to define and understand *fuzzy* comparisons to help you measure proximity between any two articles based on their attributes (also called *features*), not on the full texts.

This course introduces *document vectors*, which are sequences of numbers *representing* each text document in a corpus. These can help you to perform arithmetic on bodies of text so that you can calculate *degrees of similarity* between them. Document vectors can be created using:

1. *Counters* (of, say, specific words in a document); or
2. Sophisticated pre-trained models, which can encode some text as a finite sequence of numbers.

In this course, you will explore both of these techniques. Specifically, you will use different models to create both *sparse* and *dense* document vectors and, ultimately, calculate degrees of similarity between documents. Knowing how similar two (or more) documents are is crucial for many NLP applications, especially as you move into more advanced use cases.

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

- Create and interpret sparse document vectors using two different models
- Create and interpret dense document vectors using advanced feature engineering techniques
- Compare different document vectors to measure their similarity

Faculty Author

Oleg Melnikov, Ph.D.

Visiting Lecturer

Cornell Bowers Computing and Information Science

Cornell University

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. in Palo Alto, CA, and has decades of experience in various fields, including mathematical and statistical research, teaching, databases and software development, finance (portfolio management and security analysis), and adtech.

Dr. Melnikov's academic path began with a B.S. in Computer Science (and some years in pre-med); now, he holds Master's degrees in Computer Science (Machine Learning track) from Georgia Institute of Technology, Mathematics from UC Irvine (where he was in the Math Ph.D. program), and Statistics from Rice University, as well as an MBA from UCLA and a certificate in Quantitative Finance (MFE equivalent). Passionate about education and hard sciences, Dr. Melnikov has taught 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.

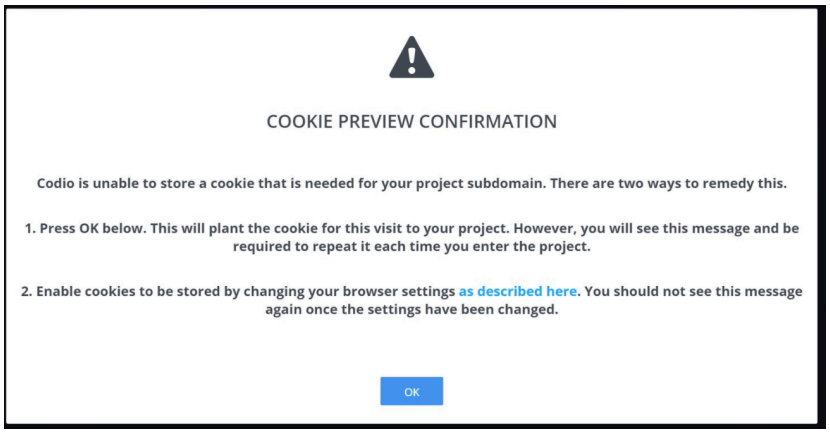
You will interact with Python throughout this course by writing and running code in workspaces hosted on Jupyter Notebooks. Our cloud-based implementation of Python and Jupyter Notebooks means that you can complete this course without installing Python on your machine.

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 full screen by clicking the full-screen button in the bottom-right corner of the workspace. The image below is an example of what the button looks like.

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



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 (aka Codio unit) that has Python and the necessary packages/dependencies pre-installed.

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

1. **Set up** 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 tool linked here to familiarize yourself with Jupyter Notebooks or to refresh your memory for how they function. You can also save it for future reference.

To download the file, click the "Pop-out" icon in the upper-right corner and open the tool in a new tab.

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 — Creating Sparse Document Vectors

In this first part of the course project, you will construct various document matrices and perform operations on their values using Pandas DataFrames.

— Course Project, Part Two — Creating Dense Document Vectors

In this part of the course project, you will use a pre-trained Word2Vec model to retrieve word embeddings (i.e., vectors) and use these to identify semantically closest neighbors. You will use the `gensim` library to complete several functions to eventually build a chain of word neighbors.

— Course Project, Part Three — Measuring Similarity Between Vectors

For the final part of the course project, you will build a document vector and identify words within the document that are semantically the closest to that vector using several different metrics.

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).
- 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 one another 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 one another, 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: Create Sparse Document Vectors From Counters

Simply by equating the strings of characters, you can easily say whether or not two documents are the same. Yet you often want to express relationships with a degree of similarity; in other words, you do not want to ask "Are these two documents the same?" but "How similar are these two documents?" Answering this question is easier with mature mathematical tools once you represent the text within the documents numerically. Enter the *document vector*, which allows you to perform arithmetic on text.

In a document vector, you can compress information and represent words with vectors that carry similar semantic information. In this module, you will construct and evaluate two kinds of vectors: the *document term matrix* (DTM), and the *term frequency-inverse document frequency* (TF-IDF) matrix, a special kind of DTM. You will also work closely with the `scikitlearn` package, which contains the tools to quickly compute a basic DTM and a more sophisticated TF-IDF DTM.

Back to Table of Contents

Watch: Operations on Vectors

You can do analyses on numbers that are not possible on text, so numerically representing text opens up many analytic possibilities. Often, the text in a document is converted into a *vector*, which is just a sequence of a few numeric values. You will then need to understand vector structures and how these are transformed in *linear algebra* operations.

Libraries/Methods

NumPy Library

`np.array`, `np.concatenate`, `np.dot`, `np.newaxis`, `np.ones`, `np.r_`, `np.zeros`, `np.ndarray.copy`, `np.ndarray.shape`, `np.ndarray.T`, `@` (same operation as `np.dot`)

Matplotlib Library

`plt.axes`, `plt.grid`, `plt.xlim`, `plt.ylim`, `plt.tight_layout`, `plt.show`

Here, Professor Melnikov discusses vector fundamentals and demonstrates how to create them in Python. He also demonstrates several arithmetic operations of vectors, including addition, subtraction, and scalar multiplication.

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

Video Transcript

A vector can be thought of as a point in Cartesian coordinates system space or an arrow from an origin, which is the head, to the point, which has a tail. In one-dimensional space of a real number line, a vector is just the number, whether positive or negative. In two-dimensional space, it has two coordinates, one for each of the axis of that coordinate system. In a three-dimensional space, a vector is represented by three coordinates, and so on. It can have many, many, many, many more dimensions.

Contrary to scalar values, vectors in two higher dimensions cannot be ordered. However, we can still compute distances between points or vectors, and that will be helpful to us. Vector representations of text can store far more information than its numeric representation of just scalar values. Basic operations on vectors include addition, subtraction, multiplication by a scalar or a number. Positive scalar multiplication changes

the vector's length or magnitude but does not change its direction. Negative scalar multiplication flips the vector about the origin.

For vectors A and B, the new vector A plus B starts from the origin and extends to the tail of B if we consider B's head placed at A's tail. The new vector A minus B is parallel to the line segment connecting the two tails A and B and has the same magnitude of the same line segment. The tail of A minus B is basically the same vector flipped 180 degrees, but it lands at the point A. You can think of all the vectors starting at the origin. Doesn't matter where we compute them or how we measure them; they should all start at the origin. So we don't care about the starting point; we only care about the point where the arrow is pointing to. In that sense, they're equivalent two points.

All the separations are element-wise and only make sense amongst same-size vectors. So you cannot add a two-dimensional vector to a three-dimensional vector; you wouldn't know what to do with that other dimension. Thus, a scalar multiplication is basically multiplying each element of the vector by that same scalar. A vector A plus B has all its elements as sums of the corresponding elements of A and of B. We will discuss vector products later, but it's a very fascinating field, multiple operations, actually, that we will make use of. Every element of a vector is indexed by a contiguous set of integers, which start from zero in NumPy array representation of a vector.

Let's look at some operations on vectors in Python code. Here we have a couple of libraries that we'll need to represent and plot vectors, and the function that will actually do the plotting and labeling of vectors in two-dimensional space or a plane. We have two vectors, a and b, declared and all these different operations plotted with that function, and some additional manipulation on the plot to make it look nicer. So vector a — all of these vectors and the vector a start at zero. Vector b starts at zero, and they basically point to this value — or to these vectors or points in space, [1, 2] for the a and [2, 1] for the b.

The scalar multiplication by a positive number 2 basically extends the vector a on the same line, in the same direction, by twice its length, and negative 0.5a will flip the a and shorten it by half. The summation of a and b will create a new vector, a plus b, where you can think of it as vector b placed to start at a and extend by the same length and in the same direction to this point, a plus b. You can also think of it as vector a placed right here at b and extending to the a plus b. The a minus b and b minus a are opposite vectors but they are 180-degree opposite; they both start at zero, and a minus b is a vector from b to a, the same length and same direction. Again, it's placed at the origin, but it's the same vector from b to a.

Code: Practice With Vectors

In the previous video, Professor Melnikov introduced some basic operations that you can do on vectors. 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 use these operations on simple vectors to verify that you understand how these operations work conceptually and in Python.

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: Operations on Matrices

Another important data structure is a matrix, which is a two-dimensional array of numbers that can be formed by stacking same-size vectors and looks like a table (or box) of numbers. In NLP, matrices are useful because they can store information from multiple documents in a single convenient structure, which means you can simultaneously operate on values from all the documents in which you're interested.

Libraries/Methods

NumPy Library

- `np.array()` object
- `.T` attribute

SciPy Library

- `csr_matrix()` object
- `csr_matrix().toarray()` method

Follow along as Professor Melnikov explains the various types of matrices then demonstrates several key operations on matrices.

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

Video Transcript

If a vector in 1D space is a one-dimensional array, a matrix is a two-dimensional structure. Think of it as a rectangular table of numbers or vector of vectors. We can stack vectors vertically or horizontally to form a matrix, which allows us to store many points in a single object and operate on all of them at once instead of one at a time. Each element of a matrix has a row and column integer index. As before, we can multiply a matrix by a scalar and we can add and subtract matrices as long as they are all of the same size.

The more we know about matrix structure, the more it is useful to us. Let's look at some structural definitions. We have a square matrix, which has the same number of rows and columns. Transpose matrix is flipped about its diagonal. Symmetric matrix is unchanged if transposed, so the elements of diagonal are the same in correspondent positions. Diagonal elements, by the way, are those that have the matching indices, so A11, A22. They're all located on a line of numbers in the matrix that goes from the top left corner to the lower right corner. All the other elements are considered to be off diagonal. A diagonal

matrix in particular has all off-diagonal values as zeros. It is always symmetric. Identity matrix is one of those diagonal matrices that has 1s on the diagonal.

Sparse matrix has mostly zero values. We don't know how many, but — there's no exact definition, but primarily zero values. The dense matrix, on the other hand, contains mostly non-zero values. Highly sparse matrices in NLP are very common and they waste storage and memory, so we often store on the row and column index of the non-zero elements and the correspondent values. SciPy Library has a special sparse matrix data structure for sparse matrices. Our typical matrix in NLP will have thousands of rows and hundreds of columns, but for now we'll look at some elementary matrices in their operations using NumPy array in our code.

We have this matrix A with values 1 and minus 2 in the first row, 3 and 4 in the second; we can print it out and display it in the output. We can multiply a matrix by a scalar 10, and that multiplies it element-wise, so every element of that matrix is scaled by 10. Finding the transpose is as easy as adding a dot T, which is the attribute of the matrix object, and it will flip the matrix around the diagonal. So minus 2 goes to the left lower corner and 3 flips around the diagonal. If you add a matrix and its transpose, you always get a symmetric matrix. The symmetry comes from the off-diagonal elements. The diagonal elements may be different, but the off-diagonal elements in corresponding positions will always be the same. You can check for the symmetry by equating the S to its transpose, and if it is symmetric, you will have true in all the other positions; true Boolean values.

The shape of a matrix or the dimensionality of a matrix could be retrieved with the shape attribute; 2 for the rows, and the second 2 is for the columns. Here's this scipy.sparse matrix format that I referred to earlier, and it basically stores the matrix in a dictionary or a list object, but it doesn't store its zero values; it doesn't store the denser presentation. Some description is contained in this three-by-three sparse matrix, where only three elements are stored, and it's 1, 3, and 4 in some positions. We can convert it back to dense format of NumPy array by using the two-array method, and the dense representation is displayed.

Back to Table of Contents

Code: Practice With Matrices

In the previous video, Professor Melnikov introduced matrices and demonstrated how you can create and manipulate them in Python with NumPy arrays. 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 manipulating and performing operations on matrices.

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: Operations on Pandas DataFrames

Pandas DataFrames are user-friendly tabular structures that store and operate on record-level data. DataFrames are useful because they are highly flexible objects with many popular attributes and built-in methods that can be applied to its values.

Libraries/Methods

Pandas Library

- `.DataFrame` creates a DataFrame object
- `.Series` creates a series object
- `.DataFrame.shape` retrieves dimensions
- `.DataFrame.groupby` prepares columns for grouped statistical summaries
- `.Series.str` access to string operations
- `.Series.str.len` vectorized length
- `.Series.mean` average values of a series

In this video, Professor Melnikov walks through Pandas DataFrame fundamentals and demonstrates how to slice DataFrames; filter rows and columns on some conditions; and compute statistics on rows, columns, and groups.

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

Video Transcript

A Pandas DataFrame is similar to a NumPy array but allows columns to be of different datatypes. One column could be numeric and another could be strings or contain strings. This is what's called a tabular format, and it's very similar to the tables in a database type of format. Many algebraic operations for the arrays and DataFrames would be the same; in fact, Pandas DataFrames use NumPy arrays underneath. Some major differences make DataFrame slower but more user friendly with plotting operations, custom row and column labels, aggregation functions, filtering functions, and many other useful operations.

Let's see some of this in action. We have this NumPy array that is created from a list of numbers from 0 to 49, and it's reshaped to have five rows and ten columns. It is then packed into a DataFrame of the same shape, and the names are being assigned to rows and columns. We have V through Z names, and A through K names for the columns. The shape of this object can be extracted through the df.shape attribute; five rows, ten columns, just like we expected. Each column can be extracted either as a series if it's

wrapped in a single set of brackets — so a series named A is extracted — or if you put double square brackets, you can extract a column and keep it in DataFrame format. Series and DataFrames are very similar, but there are some subtle differences that may be of interest later to us.

The DataFrame can also be sliced on a range of integer indices. Here we're slicing out rows 0 and 1; the 2 is left out, it's not considered, but we have to specify from zero to two rows. We can slice the columns similar way and — columns and rows and basically extract subtables or submatrices from the underlying NumPy DataFrame. We can load DataFrame with strings. So here we extract the methods for an integer 0; integer's an object, and it has methods, and all these methods are extracted. Some of them are private, some of them are public. The private ones have double underscores on either side of the name of that method. We store this all into a methods column. Likewise, we can extract the size of each method string or name and save it into a length column. This is done with the string attribute called on the column methods containing those strings. Then we have access to all string operations, including len, and len will give us the size of each. So the abs has three characters plus the four from the underscore — that's length seven — and so on.

Might be interested in the average length of the method names, and it's 8.8. All we have to do is call the length column and the mean statistics on it in the same way we can call median, mode, and many other statistical functions to evaluate the distribution, standard deviation and variance being some of those. We can create a mask. A mask is just an array of Boolean values that mark the rows of interest. So here the rows of interest are all the private names of methods. df.Methods will return the column methods. The string will allow us to call the startwith, and we can return whether every one of these methods starts with an underscore or not. All that is saved in isPrivateMask, which is another column we create in DataFrame of just the Boolean values. We can use these Boolean values or this mask to filter out all the rows that do not have private names. We have 62 rows remaining with the private names or the underscores in the name. Further, we can groupby this mask and each group — private and public names of methods — will have its own median computed. On average, the private methods will contain eight characters minus four that are underscores, so four characters on average.

Back to Table of Contents

Code: Practice With Pandas DataFrames

In the previous video, Professor Melnikov introduced Pandas DataFrame objects and demonstrated how you can create and manipulate them in Python. 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 and manipulating Pandas DataFrame objects.

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: Plotting Vectors, Matrices, and Pandas DataFrames

Sometimes you may want to visually represent complex data using plots. Use this tool to review the plotting conventions used throughout this course and reference different methods for plotting data using the Matplotlib, Seaborn, and Pandas libraries.

Please complete this activity in the course.

Back to Table of Contents

Watch: Document-Term Matrix (DTM)

Once you've preprocessed the documents you want to analyze, creating a document-term matrix (DTM) is one approach you can use to numerically represent documents. DTMs don't store information on how words and sentences co-occur within a document, so they are known as a bag-of-words type of model — the order and punctuation within each document is ignored entirely. DTMs store documents on the rows of a matrix and the frequency of specific words in each document on the columns. Since both rows and columns of a matrix can be manipulated as vectors, storing matrices in this manner opens up a wide range of analytic possibilities.

Libraries/Methods

Scikit-Learn Library

- `CounterVectorizer` object
- `.fit_transform` learns vocabulary, returns DTM
- `.get_feature_names` retrieves vocabulary

SciPy Library

- `csr_matrix().toarray()` method converts to dense matrix format

Seaborn Library

- `heatmap` plots colored grid

In this video, Professor Melnikov demonstrates how to construct DTMs and discusses how you can use DTMs to perform rough comparisons of documents.

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

Video Transcript

In preprocessing, we remove and standardize the lexicon of a corpus. Then we need to count words in each document. All counts are saved to a matrix where rows represent documents and columns represent words or terms from extracted vocabulary. Because we ignore co-occurrence of words, this model is called a bag of words, and the columns can be in any order. The value 0 in position (i, j) indicates the word or a term in column J was absent from the document in row I. We call this a document-term matrix, or DTM. Positive values of this matrix can all be 1s to indicate the presence of a word, or we can store the actual counts of words in the documents.

This matrix is commonly constructed with CountVectorizer object of Scikit-Learn library. This object can allow for some very nice pre-processing before we even generate the DTM matrix so we don't have to preprocess it as heavily or preprocess text. Once we have our document row vectors in that matrix, we can use this to compare the similarity among documents based on how many words they have in common.

Let's look at this in action. We have Scikit-Learn's object CountVectorizer, which will be doing most of the work, and Ls — or list of strings — Quote contains 14 different quotes from all around the world about a language. The CountVectorizer object is printed — or all of its arguments are printed. You can see that there are many more arguments that we could have modified or provided, one of which is token pattern, which can be used to provide a different tokenization for our texts. We fit and transform LsQuote, which basically applies the preprocessing and extracts all the vocabulary in lowercase. What is returned is this sparse matrix format we've seen earlier, which has 48 columns for the terms with the words that are extracted and 14 rows for the documents. The non-zero or positive elements of this matrix are very few or scarce, and there are 64 of them. Do this matrix format, or the sparse format, is appropriate for this type of structure. It saves us memory and storage on a computer.

The list of strings is this vocabulary object or variable and it contains all the extracted vocabulary in lowercase ordered alphabetically. We can use this later to label our columns. If you want to look at the actual values, we just need to convert this to a dense format; toarray method will do that. Most of the values are zeros, just like we expected, and some of them contain one or two positive counts. 2 indicates that this word in this column was appearing twice in the first sentence. We don't know which sentence it was or which column — which word is represented by the column until we label these columns and rows, which we can do with a DataFrame.

Here is a DataFrame where "A different language is a different vision of life" is a first label for the first row and the word "different" appears twice. It doesn't appear in anything else that is displayed currently, and we can say that this sentence is less like the other sentences. We can look at the whole matrix in a little bit more compact and visually attractive manner was a Seaborn heatmap method, which allows us to annotate the values or the cells with the numbers and this coolwarm coloring highlights high values like the number 3 here for the word "language." The C bar, the color bar, is removed from the right. This word "language" appears three times in the sentence "A special kind of beauty exists which is born in language, of language, and for language. The word

"language" appears three times. There are some other sentences or quotes with the word "language," which makes them all a little bit more similar to each other. And more similarity across different documents is driven by these other similar counts for the same words, like the word "soul" appears three times — or appears in three different documents. The word "thoughts" appears twice in different documents. We will use this later to compute and master the metric of similarity between these different documents and between words.

Back to Table of Contents

Code: Practice With a Document–Term Matrix

In the previous video, Professor Melnikov introduced the document–term matrix (DTM), which is one method of converting a group of documents into a 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 explore creating sentence vectors, turning groups of sentence vectors into DTMs, and visually comparing similarity among these sentences.

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: Term Frequency-Inverse Document Frequency (TF-IDF)

In contrast to the word counts inside a DTM, the values inside a term frequency-inverse document frequency (TF-IDF) matrix are scaled in a way that takes both the *overall word frequency* and *document-specific word frequency* into account, such that:

1. Words that are found in every document have relatively low weights even if they are found frequently, whereas
2. Words that are found frequently in only a few documents have relatively high weights.

Here, Professor Melnikov defines TF-IDF matrices then discusses why they are favorable for comparing documents as well as how to calculate them for words in a document within a larger corpus.

Video Transcript

A big disadvantage of document-term matrix, or DTM, is that generic words tend to have high counts, because they are nonspecific and common to too many documents. Of course, we could add these to the stopword list, but this would require a tedious and regular management of the stopword list. Alternatively, term frequency - inverse document frequency, or TF-IDF, can automatically assign low weights to generic and infrequent words. It also assigns high weights to frequent terms or words found in fewer documents. So if we have books on biology and NLP, stopwords common to all of these documents would have low weights. But biology terms that appear in biology books only would have higher weights because there are specific to that corpus.

Term frequency or TF component of the formula is the count of the term in the specific document. It's specific to the vocabulary term and to the document. Inverse document frequency or IDF component is the inverse of the document frequency for each term. To compute it, we divide the total number of documents by document frequency for each term. Often we apply a logarithmic smoothing. Log function shrinks counts towards zero and it does so more aggressively for very large or extreme counts.

TF-IDF is also a document-term matrix, or DTM. In that sense, it's still is troubled by high sparsity or lots of zeros and large vocabulary. Now, the TF-IDF does not remove words from vocabulary, but we can detect stopwords by thresholding the weights in TF-IDF matrix.

Back to Table of Contents

Watch: Interpreting a TF-IDF Matrix

The ability of TF-IDF matrices to quantitatively define whether a word is "common" or "rare" within a corpus allows for more meaningful comparisons of documents within that corpus.

Libraries/Methods

`scikitlearn` **Library**

- `tfidfTransformer()` object
- `.fit_transform()`
- `get_feature_names()`

In this video, Professor Melnikov demonstrates how to construct a TF-IDF matrix from a collection of documents using the `scikitlearn` library then discusses how you can interpret TF-IDF matrices.

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

Video Transcript

Now that we know some theory about TF-IDF, we can learn to apply it to documents with Scikit-Learn library and interpret the weights that result from this library. Let's look at the code. Scikit-Learn gives us a couple of objects we can use to create a TF-IDF transformer, a TF-IDF object. We have this CountVectorizer that we used before and it counts words in the documents or across different documents, and then TF-IDF will transform it to different weights. The counts will be transformed to weights. The zeros will remain zeros, unchanged.

All those different quotations, famous quotations about language, are in our corpus; there are 14 of them. And because "language" appears in many of these sentences, it will likely be a stopword. It's unimportant in identifying different sentences. The CountVectorizer object takes some parameters where you can control the preprocessing. The English stopword list is applied, and the stopwords — generic stopwords are removed. "Language" is not one of them, but "he," "he's," "is," "I," and others, if appear they will be removed. The words are a lowercase, and you can provide a preprocessor later, a more complex preprocessor, to stem or lemmatize the words or de-accent them and so on. The fit-transform will actually do the counting and parse out the vocabulary. The vocabulary will be stored and accessible later.

For now, we have a result sparse matrix with 14 rows, one for every sentence or a document, and 48 columns, one for each term or a word. This is a document-term matrix, 64 values are non-zero, they are counts, and we can convert them to weights. There is a vocabulary that results from the parsing. None of the stop or generic stopwords are here, and in this next cell, what we're doing is applying TF-IDF transformer to the DT sparse matrix that we created earlier. And this will basically weigh each count according to appearances of that word in the document and all the documents as we discussed; TF-IDF does.

If we look at the result and interpret what we're actually seeing, there are some highlighted nicely, red color, for the very high weights. The weights vary between 0 and 1, so 0.9 is very high for the word "change," and that's because "change" appears twice to more frequently in a single document and doesn't appear anywhere else, and the document itself or the sentence itself is rather short. The word "change" has a much greater weight in it and not anywhere else. Same goes for the word "knowledge," for the word "limits," and so on.

Notice that some of these words can be combined, like the "knowledge" and "knows" may be considered the same synonyms and structurally are more morphologically similar, have the same root, same meaning, so in preprocessing, we might want to account for this. Same goes for the "language" and "languages"; the plural and singular might be considered to be the same word. So we would have better, finer-tuned weights, results in the matrix, after this type of preprocessing. The word "language" unsurprisingly has lower weights — 0.17, 0.18 — that we see across all these different documents, and that's because it appears throughout — pretty much in every quote. In some sentences, the word "languages" appears Instead of "language," and that could be improved if we lemmatized first.

There are some words where the semantic meaning is similar but the words are very different, so lemmatization will not help; like the word "wisdom" and word "knowledge" are semantically similar. And we'll look at techniques later in the course that will handle this situation. If we look at this matrix, we can identify stopwords — new stopwords that are not important in distinguishing these documents, one of them being the world "language" that has very low weight. But there are other words, like "understand" or "change," that have very high weights and are not considered to be stopwords. They would identify a particular document uniquely, so we want those words to stay in the corpus or in vocabulary.

Code: Interpret a TF-IDF Matrix

In the previous video, Professor Melnikov introduced ways to build and interpret a TF-IDF matrix, which you can use instead of a document-term 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 creating and interpreting TF-IDF matrices.

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: Thresholding a TF-IDF Matrix

In addition to its usefulness in comparing documents, the TF-IDF matrix can also be used to automatically identify important words or stopwords in a corpus. Stopwords greatly depend on the context, and manually identifying these words for specific contexts can be time consuming.

Libraries/Methods

`pandas` **Library**

`DataFrame()` object

- `.max()`
- `.columns['col_name']`
- `.replace()`

In this video, Professor Melnikov walks through how to threshold the weights within a TF-IDF matrix and use these thresholds to automatically highlight terms based on either their importance or lack thereof.

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

Video Transcript

Now that we have developed familiarity with TF-IDF matrix, let's look at how it can be used to identify important words and stopwords automatically. We'll do this in code. So we are extracting TfidfVectorizer object that will help us to push the whole pipeline: the preprocessing, the counting of the words, and converting counts to weights. We have 14 different phrases from famous people all around the world about language and the word "language" appears in many of them. So the "language" word will naturally be a stopword or it should be detected as a stopword.

As we run this object, create this different object, we are not passing stopwords list anymore. We're still lowercasing, and we want to automatically detect the stopwords that are specific to this corpus. The transform will apply — the preprocessing will count the words and we'll apply the weighting function, TF-IDF weighting function. The result is 14 rows, one for each sentence, and 90 columns, which now includes the stopword columns. It's about twice the size of what we had before, which was 48. We have 136 non-zero elements with weights, whereas we had about 60 of them before.

We can present it in a very nice, colorful DataFrame format with a library called Seaborn, which displays it in a heatmap with a coolwarm coloring where the larger numbers, which themselves vary between 0 and 1, are presented in a reddish color and the blue indicates closer to zero. So at the bottom here we have labels for the columns. This is now a transpose version of DTM, or flipped around its diagonal, so it's a TDM or term-document instead of document-term matrix. The documents are the columns, the terms are the rows. We are looking for words that have high weights or low weights. Those are important and unimportant words.

As you can guess, it will require some sort of thresholding. So we can put a threshold on the values. Let's say we can take the word "different" and look at all of its weight and it has only one. We can take the maximum weight and threshold it; let's say a threshold of 0.5. This word will be highlighted, many of the other words will drop out. The word "change" will be highlighted as well. So these are important words that are important to at least one document. The unimportant words will have reasonably consistently low weights, like the word "language," except the word "language" appears to be reasonably important for a document where it appears three times.

So what do we do? If we just take the minimum, that might cut out some other words that are important throughout but may not be important in one case, Like the word "off" is reasonably important, but in this one case it's 0.17 for document. What we can do is we can ignore the zeros, compute the mean of this value. So this is one technique; there are many others. Computing the mean of these values and threshold on that. So let's see how it's done. So first, the important mask vector or series is calculated. We're taking the maximum across all these different rows, all the different columns for each row, and thresholding at 0.5. So if it's greater than 0.5, it will come back with a true for the word "yours." Then we take this mask and apply it to the TF-IDF columns, which contains different words in the vocabulary; all the words in vocabulary here. Only the ones that have true value in the mask will come back to us. We can see that these words are important; even though they are on a generic list of stopwords, they are not stopwords; they are very important to this corpus.

At the same time, we can identify the stopwords for this specific corpus by first replacing all the values — all the zero values with "none" or "not a number." This "not a number" is not used in mean; it's ignored by a mean function. Then we are computing the mean or the average, the simple arithmetic average, across all the columns for each of the rows. Here are the average weights. We can threshold those by a number — let's say 0.25 — or

we can pick some percentile if we want it to be more robust to different types of means. This is a similar type of masking vector or series that we can apply to TF-IDF columns again and highlight or return the words that are stopwords. For language, unsurprisingly "here," "is," and "and" I remember appeared multiple times as well.

Back to Table of Contents

Code: Threshold a TF–IDF Matrix

In the previous video, Professor Melnikov introduced ways to threshold a TF–IDF matrix to find both the most important and least important words in the corpus in which you're interested. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video, paying close attention to what "important" means in the context of your analysis. In the "Optional Practice" section of this exercise, you will practice thresholding TF–IDF matrices.

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: Create Matrices From Text

Before analyzing a corpus using NLP techniques, you may first want to represent your text's contents within a matrix. Use this tool to review the types of matrices you can create from text, and refer to it when creating your own matrices from a corpus.

Please complete this activity in the course.

Back to Table of Contents

Assignment: Course Project, Part One — Creating Sparse Document Vectors

In Part One of the course project, you will construct various document matrices and perform operations on their values using Pandas DataFrames. 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.

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 up to three hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Create Sparse Document Vectors From Counters

You often want to compare documents by expressing their degree of similarity, and one relatively simple way of achieving this is to transform text into numeric vectors, then count (or somehow measure) the overlapping tokens. In this module, we constructed and evaluated two kinds of vectors: the document term matrix (DTM) and the term frequency-inverse document frequency (TF-IDF) matrix. We did this by working closely with the `scikitlearn` package.

Next, we will extend this idea to represent words as dense vectors, which we can aggregate to represent sentences as dense vectors; this allows us to measure similarity between any two sentences, regardless of whether they have the same words or not. In fact, we can even measure similarity between words and sentences and in different languages. All that is possible when we convert our text to mathematical vector spaces.

Back to Table of Contents

Module Introduction: Create Dense Document Vectors From Pre-Trained Models

In the previous module, you used counts and weighted counts to create sparse document vectors (both a DTM and a TF-IDF). There is another way to build vectors, which uses the Word2Vec model. This model does not count words in a given document but uses predictive deep learning on very large corpora to create representations of words that capture contextual and semantic similarity. The resulting numeric vectors are dense rather than sparse and consequently have much lower dimensionality.

Thus, while a corpus can have a vocabulary of a million meaningful words (in their lemmas), your vectors will be anywhere from 50 to 300 dimensions, which is a much lower representation. Since there will be practically no zeros in these vectors, you will no longer need SciPy's sparse matrices to represent your abstract matrices; you can just use the well-familiar NumPy arrays.

In this module, you will explore dense document vectors, and explore the Word2Vec model. Additionally, you will use a new package, `gensim`, as you build and train a model to create your own dense document vector.

Back to Table of Contents

Watch: Dense Matrices and Word2Vec

Unlike the DTMs that are characterized by high sparsity, vectors from a Word2Vec model are dense representations of text that capture the syntactic and semantic relationships of words in a corpus across many dimensions.

Libraries/Methods

Gensim Library

`KeyedVectors` object

- `.load_word2vec_format()`
- `.word_vec()`
- `.vector_size`

In this video, Professor Melnikov introduces the Word2Vec model and discusses its advantages and limitations. He then demonstrates how to load a trained Word2Vec model and observe the dense vectors for several words within its vocabulary.

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

Video Transcript

Sparse vector representations are computationally inefficient. Recently, a dense representation of text have been developed where text is encoded into 50 to 500 dimensions instead of hundreds of thousands of dimensions required in sparse representation of DTM and TF-IDF. Much of it started around 2013 when Thomas Mikolov, employed by Google at that time, was able to train a Word2Vec model on almost 2 billion words. This was a simple neural network which learned associations between neighboring words in its hidden layer of coefficients. These coefficients were precisely the numeric vectors for words fed through the model. It was surprising to find out that the model captured syntactic and semantic relationships among words into dimensions of the vectors. These relationships relate countries to their capitals, currencies, nationalities. They also associate various forms of verbs, nouns, adjectives, and other morphological structures or derivatives. It understands synonyms, antonyms, comparative word forms like "great" or "greater," and much more.

The biggest limitations of Word2Vec model were slow sequential training, limited vocabulary, and static vectors regardless of the word's meaning in a sentence. For example, if a word, such as my last name, was not part of the training corpus, there would

be no way to get a vector for that word. Also the word "jaguar," which would be defined by the same trend vector regardless of the context it appears in but it could mean car in one sentence and it could mean an animal in another. However, both senses — car and animal — of this "jaguar" word would be captured by a single vector. And there would be dimensions that are presenting more of an animalness of that word and more of a carness of that word. Since then, the models have improved dramatically.

But for now, let's see how the word vectors are loaded with Gensim package in the code. We need to download the Gensim library and — Gensim library trained Word2Vec model, and this could be brought in from the GitHub, from the RaRe Technologies. It's the smallest models that they have of 50-dimensional vectors. We can even see the size of this model, which is roughly 17 megabytes. It's a .gz or zipped format, and we can uncompress it, actually look inside of this trained model. It's just a text file. It's a text file that indicates on the first line how many word vectors there are in the file — there are 400,000 — and these are 50-dimensional words. This word "the" that appears in the first line is the article word that has this following representation of numbers or the vector, which is 50-dimensional. So if we scroll all the way to the right, this is the 50th — or 49th if we start from 0 position; all the coefficients that was trained in the neural network in the Word2Vec — or actually GloVe model, which is very similar. There are 400,000 of these if we scroll all the way to the bottom; 400 001 was the first line.

And we can search for a particular word, like "cornell," and find its vector is a vector starting with negative 0.929. There are other words that are also here and are represented by a vector. Loading this model or this 70-megabyte file — it's actually unzipped 170 megabytes — takes a little bit of time, so we preloaded this for you. When you print it out, it just says it's a Gensim model of keyed vectors. It's basically a dictionary with additional features or attributes and methods that we can use, one of which is extraction of a vector for a particular given word. Word_vec method will extract a vector negative 0.929 that we just saw in a text file. All it does, it just pulls that line out of the text file. We can actually refer to the object wv, the model object, as we refer to a typical dictionary and extract the vector coefficients in the same way. There is another attribute that is convenient to use; you can count number of coefficients here and derive 50, but you can also extract vector_size and that will tell you the dimensionality of a vector.

Back to Table of Contents

Code: Practice With Dense Matrices and Word2Vec

In the previous video, Professor Melnikov introduced the Word2Vec model and demonstrated how to load, manipulate, and interpret a `gensim`-trained Word2Vec model. 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 the model to make some interpretations about word similarity.

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: Word2Vec Vectors

As you've begun to examine, the information embedded inside the dimensions of Word2Vec matrices can be used to identify syntactic and semantic relationships among words.

Here, Professor Melnikov discusses how you can glean information about what the dimensions of a Word2Vec matrix represent by closely examining the matrix.

Libraries/Methods

`try` and `except`

`gensim` **Library**

`KeyedVectors` object

- `.load_word2vec_format()`
- `.vectors()`
- `.vocab`
- `.keys()`

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

Video Transcript

Gensim package is rich with functionality, not just for Word2Vec but for several other word-embedding models such as GloVe and FastText. It offers a standardized interface for extracting Word vectors, computing similarities, searching for vocabulary — or searching for words in vocabulary, and much more. Let's try some of this in code. We are, as usual, loading this GloVe model, which is very similar to Word2Vec and it's trained on Wikipedia. This is the smallest model that is out there, typically found with 50-dimensional vectors and just 400,000 lowercased words. We are loading the model into the wv object. This is the returned result; it's loaded as the keyed vector, so basically a dictionary with additional attributes and methods.

Notice that because the words in this dictionary — or the keys in this dictionary are all lowercased, if you tried to search for uppercase word and extract its vector, you will get an error message; we're capturing this error message with a try-and-except to make sure it doesn't crush the code. But the error is essentially "word 'cornell' is not in vocabulary." The extraction could be improved a little bit where we can wrap it into your function, and if the word is in the dictionary, we would extract the word, just like we would extract value based on the key from any other dictionary. Or if it's not there, we could return the zero-vector,

50-dimensional vector, and it can be as easy as taking an arbitrary vector — let's say the first one, the zero vector — and multiplying all its coefficients by 0, basically zeroing them out. You can also use NumPy 0 to create a 50-dimensional vector of zeros.

The type command will tell us what the type of the wv.vocab attribute is, and it's a dictionary. We can extract its keys. The keys are precisely the vocabulary in the order that we've seen them in a text file, starting with "the," then a comma, period, and so on; 400,000 words that we put double spaces in between. We can also look at the vectors that represent this vocabulary. Every row here, just like in the text file, represents a word. This first row is for the word "the," and so on. There are 50 dimensions and 50 columns for each vector. No one really knows what these columns represent; it's an open area of research. But they are helpful in identifying the similarities between words, as we will demonstrate later.

The NumPy array could be nicely wrapped into a DataFrame where each row will get a word identifier or identification or label, and the columns will be numbered from 0 to 49 to indicate 50-dimensional vectors. There are 400,000 of this; they're nicely formatted, and it's a bit difficult to tell where the numbers relate and where they do not. We can further just focus on a few words, such as "cornell," "graduate," "university," "professor," "jazz," and extract just the vectors for these words and present them with the Seaborn heatmap in this nicely colored background table.

Now we can see that the words, such as "cornell," "graduate," "university," and "professor," have high coefficients in some of the columns, and these coefficients are different from "jazz," "music," and "wave." So the columns 37 and 38 stand out. This has some dimension or dimensions that relate to university or relate to education or relate to Cornell and they are highlighting that fact by values different from the values of the music-related concepts. Some music-related dimensions might be column 9, where the coefficients are high for the music words. There are some other dimensions which may indicate different properties or different meanings, different senses for these different words, and it's very interesting to investigate.

Back to Table of Contents

Code: Practice With Word2Vec Vectors

In the previous video, Professor Melnikov wrote a function to improve error handling for words that aren't present in the Word2Vec model and discussed how you can gain insight into what the dimensions of a Word2Vec vector signify. 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 interpreting these matrices and using them to understand the dimensions.

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: Arithmetic With Word Vectors

What do you get when you add and subtract word vectors? Since word vectors capture the semantic and syntactic meanings of words from the main corpus of a Word2Vec model across a variety of dimensions, you'll often retrieve a word that makes sense in the context of the arithmetic you've done.

Libraries/Methods

`gensim` **Library**

`KeyedVectors` object

- `.load_word2vec_format()`
- `.most_similar()`
- `.most_similar_to_given()`
- `.similarity()`

`scikitlearn` **Library**

`PCA()` object

- `.fit_transform()`

For example, what is `king + woman - man`? As you'll see in this video, it is `queen` based on the GloVe model vector embeddings and operations in the `gensim` library. By performing arithmetic using the word's vector embeddings, it becomes possible to identify words that are semantically similar to this expression.

Here, Professor Melnikov demonstrates how to do this using functions from the `gensim` library and highlights the tremendous power of the word embeddings.

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

Video Transcript

We're now ready to apply vector algebra we learned earlier to word vectors. We do not need to explicitly compute — add or subtract vectors — because Gensim library provides this functionality for us. But we will explore how it's done — how it can be done and what other methods are available for us to operate on vectors. Let's look at the code. Here we have the GloVe word to vector model loaded. Again, it has 400,000 words lowercase and 50-dimensional vectors representing each one of them. We can load it as usual into wv object in Python. This is mostly a dictionary — a Python dictionary with some additional methods and attributes.

One of these methods is most_similar, and it'll take a word, "cornell," and return topn — top number —10; in this case, words that are most similar. By "most similar," we mean they have the greatest cosine similarity. We'll learn about this later; it's a number between negative 1 and 1 indicating how similar the two words are. It's a pairwise metric, so it needs to take two words; one of them is "cornell" and the other one is — every word in the vocabulary is tried; every one of the 400,000 words is tried in 400,000 computations. Then they're sorted or ordered by this similarity and the top 10 are returned.

We can limit this computation to just a few selected words, and if we want to know which of the words in this given list is most similar to the word "cornell," we can do the most_similar_to_given and provide a list of words. The word "professor" is returned; we don't see the similarity score that was used to bring this word up to top word on that list. But here's a way where we can actually iterate over the words in the list we have, list of string words, via list comprehension, and compute similarity on the "cornell" and each one of those words. At the end, we apply a sorted method on the similarities that is returned as part of each one of these tuples, and we order by the similarity to "cornell." The "cornell" and "cornell" have a similarity of 1, of course, but the next one up is "professor," as we've seen earlier.

There's a very nice way to display a 50-dimensional vector in two-dimensional plane. For that, we use something called principal component analysis, which is the projection of this word vector's multidimensional space to a two-dimensional surface. We'll learn about PCA later, but for now, what it does, it allows us to plot the word vectors, or their two-dimensional presentation, and evaluate how close the words are based on the distance between their vectors. So the education-related or university-related vectors are all clustered together, which makes sense. Same goes for the "music," "guitar," "concert." Notice that in the vertical dimension, the university-related vectors are similar to music. If you project all of this to the left on this vertical line, they are similar, but they are different in this horizontal space.

We can do more. The most_similar method takes positive as a list of words that we want to — for which we want to add their vectors together. And the "negative," it's not a negative word; it's a negative operation. So we will take the "man" vector or vector representing "man" and subtract it from the vector that arises from adding vectors for "king" and "woman." In some sense, this is an operation of "king" vector minus "man" vector plus a "woman" vector. The resulting vector may not be in the trained model. But

what we will do next is extract the word that has the closest vector to the resulting vector and compute the similarity. So the word "queen," which makes sense, has similarity of 0.85.

We can do more. "Actor" plus "woman" minus "man" would give us actress. Another way to state it is Obama to Clinton is as Reagan to Bush is found to be the closest result of this comparison. Finally, we can find the word that is least related, related to the list of words that we have. "Breakfast," "dinner," and "lunch" are all similar in a sense, but "milk" doesn't seem to fit in as much. We can split this sentence and use the doesn't_match method to find the word that is least sensible to this list.

Back to Table of Contents

Code: Work With Word Vector Arithmetic

In the previous video, Professor Melnikov demonstrated how to perform operations on word vectors using the `gensim` library. 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 performing operations on word vectors to answer questions about word meaning and similarity.

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: How Word2Vec Is Trained

To understand the capabilities and limitations of Word2Vec, you need to understand how the model is trained. In this video, Professor Melnikov discusses the two algorithms that were originally used to train this model, walks through model hyperparameters, and discusses benchmarks for evaluating results.

Video Transcript

Believe it or not, the biggest challenge in training Word2Vec model was not the neural network architecture; it was the difficulty of scaling the training of billions of words sequentially as they are fed through the model one by one. This is a notoriously very slow situation. Several novel and clever sampling methods made it possible. Two algorithms were considered for the training — for the sampling. Each one of them required some sort of sliding window that would go over the sequence of words. In this window, say of size 5, the center word is called a target word and the surrounding words are called context words.

In the continuous bag of words, or CBOW model, the algorithm tries to predict the target word from the surrounding words as soon as there is a relationship between words that are in the window. In the skip-gram algorithm, the model tries to do the opposite: It predicts the surrounding or the context words, given the target word, which seems to be a more difficult problem but turns out to give better results and trains faster. Whenever the model makes an incorrect prediction, the model coefficients are adjusted accordingly so as to lower the error the next time these words appear in the window. Once the training is finished, the trained coefficients from this model — either one of these two algorithms — are precisely the word vectors we use.

Before the training begins, the scientist needs to specify the hyperparameters for the model. They are also called tuning parameters. They include the window size or the word vector dimensionality or the learning rate with which the coefficients are updated or the number of iterations over the input corpus or many other ones. These are nontrivial and typically require experimentation to identify suitable parameters. For example, nobody really knows whether a sliding window of size 5 or 50 or 500 needs to be used or which one is more appropriate, or whether the vector of size 10, or of length 100 or 1,000 is better suited for a particular corpus. Some intuition, however, can be drawn from the size and

complexity of the corpus itself. For example, vectors might require a double length if two languages are considered for training. This, of course, depends on some intuition that you might develop with a single-language vector length.

The others also need to decide on the preprocessing and corpus cleaning methods. It might lowercase or leave the case intact, or they might remove punctuation or leave the punctuation there or do any de-accenting and so on. Finally, the model results are evaluated and compared against the latest benchmarks in a particular NLP task, such as question-and-answer task. For example, quantitative, semantic, and syntactic accuracy needs to be evaluated against other leading neural network and classical NLP models to determine whether the model has achieved its potential and can be used in the real world.

Back to Table of Contents

Quiz: Model Architecture Basics

In this quiz, you will test your conceptual understanding of training Word2Vec models. As you select responses, keep 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: The Long Tail Problem for Out-of-Vocabulary Words

One way to reduce the training size of a Word2Vec model is to remove infrequent words that have insufficient information for learning their vector embeddings. Other approaches, such as lowercasing and lemmatization, can help increase count frequencies for words that appear in different casing or in different morphological forms; e.g., "Happy," "happy," "happiness."

Libraries/Methods

`collections` **Library**

`Counter()` object

- `.most_common()`

`pandas` **Library**

`DataFrame()` object

- `.set_index()`
- `.freq`
- `.count_values()`

Here, Professor Melnikov plots word frequencies from in the book "Alice in Wonderland" and discusses where stopwords and infrequent words are located on the frequency of word distribution. He also discusses why infrequent words can be a challenge for interpreting the semantic and syntactic meaning of words in a corpus.

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

Video Transcript

Think about the distribution of words in a document. If we order words by their frequency, we might observe the head of the distribution with 20 percent of vocabulary making up 80 percent of text. The long tail captures the remaining 80 percent of vocabulary with rare appearances. Infrequent words express few associations with other words and result in poor quality word vectors. We might drop rare words out of vocabulary and save some space in Word2Vec model. For example, any word with frequency less than six might be dropped out due to insufficient information about its co-occurrence with other words.

Let's build this distribution in our code. We are loading Gutenberg's "Alice in Wonderland" and looking at just the top 20 tokens in the text in the order that they appear. "Alice's Adventures in Wonderland" by Lewis Carroll, 1865. We can use the collections.Counter object with the list passed to it to get the most common — or the distribution of the

words starting with the most frequent and in decreasing order of frequency. So the period appears almost 2,000 times followed by single quote and then the word "the," which would probably be removed as a stopword, and and so on. We can look at the same information in the DataFrame, which is little bit nicer, more presentable, and we can scroll all the way to the right and look at the tail of this distribution with the word "THEIR" and "happy" appearing just once in the whole text. "Their," of course, might appear more times in a different capitalization. For lowercase, our words — "THEIR" will be collapsed with another "their" and the count will go up. So the word vector would be more meaningful. But the word "happy" is problematic. Unless there was a — "happy" was upper capitalization, like a title word, this is by the long tail that we do not want to have in the training of word vectors.

Here's how the distribution looks like, where most of the words that we have — almost 3,000 of them — are appearing only once in the text. Just a few words appear an enormous amount of times; almost 2,000. Let's do a bit of preprocessing. What we have here is a list comprehension where we iterate through every single word in the Ls, in list of words, that we pulled out of the "Alice in Wonderland" text. We're only keeping the ones that are longer than three characters and the ones that have only letters in the word. Keep that in mind because some of the words with dashes will be dropped out and they might be legitimate words. We're lowercasing and placing everything back into LsWords2, and these are the first 20 words that we're observing all lowercase and much cleaner than the previous set.

We are further creating the similar object, a list of tuples with a word from the vocabulary. It's only a unique version of that word and the frequencies that is corresponding to that word in the document. So the word "said" appears 462 times, "alice" appears almost 400 times — this is lowercased — and so on. When presented in the DataFrame format with the word and frequency as the indices of these two rows and the first row word is set as an index — so it's a little bit nicer and easier to observe — we have almost 2,500 words; fewer than 3,000 that we had before. The word "happy" is still here, so there is no other capitalized version of the word "happy." There might be a word "happiness," but those are two different words that would probably benefit from lemmatization. This is still long tail and this is still problematic.

Let's see how many words appear at least six times. We have this masking or marking vector which marks old words that appear more than six times. We can actually use that in the picture and put a threshold of word or words that are most frequent and less

frequent with five or fewer appearances. It's about 20/80; just like I've said before, 20 percent of words appear six times or more. This is what we would consider a head of this distribution and the rest is the tail. This tail is still problematic, but if we have a larger text like the Wikipedia corpus or if we add other books, the word "happy" would appear more frequently and would have a more meaningful vector because it appears in different contexts and we can get — or we can learn its meaning or its sense. Here the two words that are problematic for this particular training corpus: "happy" appears only once and the word "sad" appears no times whatsoever. This would be out of vocabulary, or we would call them OOV; out-of-vocabulary words.

Back to Table of Contents

Code: Address the Long Tail Problem

In the previous video, Professor Melnikov introduced the problem of long tail distribution of word counts in NLP. 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 explore word frequency distributions.

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: Generating Subwords With FastText

Libraries/Methods

`nltk` **Library**

`nltk.ngrams()`

As you discovered in the previous video, infrequent words create a real problem in the original Word2Vec models. The FastText model is a newer type of Word2Vec model that can assign a vector embedding to almost any word by training on subwords (i.e., n-grams generated from a character sliding window) of words. The vectors that represent these subwords can then be aggregated to form the vector of the original word or be combined with other subwords to create new words and vector representations.

Follow along as Professor Melnikov discusses how the FastText model works, the challenges it overcomes, and how it changes the word frequency distribution.

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

Video Transcript

Long tail distributions are problematic in NLP. However, we might try to leverage the commonality among words. Many words in vocabulary have common word parts, such as subwords or n-grams, which include word roots and affixes. This words should be similar in some way. For example, if we train a vector for "Cornell," then it should be similar to the vector for "eCornell," which might be absent from the training corpus. If we train a vector for "learn," "ing," and "ed," then we can sum or average these vectors to derive vectors for "learning" or "learned." We can even derive a reasonable sum vector for the phrase "eCornell learning." This is precisely what FastText model does: It decomposes words into all possible 3-grams, 4-grams, 5-grams, and 6-grams of characters, and trains vectors for n-grams as well as the word itself. This allows recovering average vector for nearly any word of three or more characters, even if the word was never part of the original training vocabulary; so-called out-of-vocabulary words. it can have a sensible word vector presentation. For example, a query for a word most similar to my last name, Melnikov, yields other Russian last names and concepts.

Let's evaluate long tail for a distribution of subword frequencies in code. We are basically partitioning a word "alice" through this function into 3-grams. As a result, we get a list of substrings that we can generate from word "alice"; we can pass any other word into it. There's a function in NLTKs which will do just that, except it returns a generator. Generator can only be passed over once. It's not a list; you can only go from start to end one time. You can wrap it into a list and then actually observe the contents of the generator. They are a list of tuples of single characters which can be joined together to generate the same result we've seen before, so 3-grams for the word "alice." We can then use it in a function that will take a word and will partition it into n-grams starting from the nmin and ending with nmax. This will return several different lists for different calls of the function with all possible n-grams in that range. The way it does it, it's just a loop that iterates for the n-values from nmin to nmax and calls the function, which is fairly fast because it's internally built in with the generator, which are fast components of Python.

There's another wrapper function for that, which will take a list of words and generate n-grams for that list and pack it all as a single list of all n-grams for those words. This now can be used for this "Alice in Wonderland" document or text, book, which we clean a little bit by removing all words that are too short; maybe fewer than three characters. We are getting rid of anything that contains dashes, underscores, numbers, punctuations, and we'll lowercase all the words. Now we have this list of the original words of about 14,000 words and a list of all the n-grams generated from those words, which is about 117,000. They contain duplicates so if you want to get rid of the duplicates and look at the vocabulary of words and vocabulary of n-grams or the subwords, the subwords is pretty large, but that is not a problem here. What's important is that the subwords, even though there are many more of them, are now appearing more frequently in the text. We can have a subword appearing more than six times, and that's meaningful for a training of a vector in a FastText model.

Here's a distribution of these subwords. We can see that previously we had 20 percent of words with frequency six or higher. Now we're having 28 percent of subwords in the head of the distribution. So the long tail has shrunk in comparison or in a percentage basis and we can have — these subwords have meaningful vectors that can add up into a word vector that the subwords came from.

Back to Table of Contents

Code: Generate Subwords in FastText

In the previous video, Professor Melnikov introduced the FastText model and discussed how you can use it to build vectors even for infrequent words because it trains for subword vectors, which are more common than full 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 using the model to find subwords.

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 — Creating Dense Document Vectors

In Part Two of the course project, you will use a pre-trained Word2Vec model to retrieve word embeddings (i.e., vectors) and use these to identify semantically closest neighbors. You will use the `gensim` library to complete several functions to eventually build a chain of word neighbors. 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.

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 up to three hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Create Dense Document Vectors From Pre-Trained Models

In this module, you explored dense document vectors, specifically the Word2Vec model. This technique has a much lower dimensionality than the DTM and TF-IDF techniques you explored earlier and utilizes deep learning rather than counters. You also built and trained a model to create your own dense document vector.

A key takeaway is that while Word2Vec is a breakthrough technology, it has a very *limited vocabulary* and is therefore poorly suited for a production environment. FastText, on the other hand, has potentially *unlimited vocabulary* because it uses *subword* vectors to reconstruct the vectors of original words. Be mindful that FastText is a large file, reaching 10 GB for larger models, so that much memory is needed to host this model in a computer.

Later we will see smaller (a few hundred megabytes) and more flexible models capable of creating accurate *sentence* vectors regardless of the language (English, Chinese, Russian, etc.).

Back to Table of Contents

Module Introduction: Measure Similarity Between Document Vectors

So far in this course, you have created both sparse and dense document vectors, and you compared different models to examine the size of the vectors and observe how dense vectors capture semantic information. To express the similarity between document vectors, we can use several different metrics that result in a numeric similarity score.

Higher similarity measures imply greater similarity between paired vectors or objects. Metrics can be standardized in different ways to ease their interpretation. For example, dot product is not standardized and can result in any real value. Cosine similarity, however, is standardized to be in [-1, 1] interval, which makes it far easier to understand. Two vectors with a cosine similarity 1 are perfectly similar and two vectors with a cosine similarity of -1 are perfectly dissimilar.

In this module, you will discover much more about these metrics and measure similarity across document vectors. To do this, you will explore similarity and distance metrics, interpret several similarity and dissimilarity metrics (such as a distance), and work with both TF-IDF and FastText to build sentence vectors and find similar documents.

Back to Table of Contents

Watch: Similarity

Similarity is a numeric *measure of likeness* between two objects; e.g., words, phrases, documents, vectors. This measure is used for a variety of purposes, including identifying related documents, comparing DNA strings, and finding synonyms or antonyms of words. In this video, Professor Melnikov introduces several similarity metrics and their antitheses: distance metrics.

Video Transcript

Similarity is a numeric measure of likeness among two objects. In NLP, we compute similarity among words, paragraphs, documents, or any two sequence of elements, such as DNA sequences comprised of ordered nucleotides, ACTG, and so on. No similarity metric is perfect, but each has its own purpose and advantages. Typically, we use them to identify related documents; to query related relevant web pages; to compare viral DNA strings; to find synonyms, antonyms, words that rhyme; and much more. A distance is opposite to similarity. Higher distance implies lower similarity and vice versa. Often, we can derive one from another.

To improve interpretation of similarity, we often standardize it to be in some finite range or to be a fraction of some base number. For example, correlation and cosine similarity are in the interval minus 1 to 1. While Jaccard similarity and Hamming distance can be scaled to be in a range of 0 to 1. Thus, given the similarity of 0.95, we know the documents are highly similar. On the other hand, non-standardized similarities, such as dot product or covariance, are preferred in the algorithms because of simpler and faster computation but are less interpretable to humans.

Similarity metrics vary in the way they detect pattern differences and in the inputs they take. Binary similarity is the most generic and returns True or 1 if two input objects are matching exactly and False or 0 otherwise. Other similarities can measure the structural differences of inputs, which can be arbitrary sets or strings of varying length or even numeric vectors. In this module, we focus on vector similarity and learn to measure the degree of semantic proximity among words and documents. In its simplest form, documents with higher match count of important words are closer to each other in vector space built from a document-term matrix we've seen earlier. A more precise and robust

measure can be drawn from word and document vectors, which are built to capture semantic and syntactic representation.

Back to Table of Contents

Watch: Similarity and Distance Metrics

There are a wide range of similarity and distance metrics. In this video, Professor Melnikov discusses how the dot product can be used as a measurement of similarity and elaborates on its normalized counterpart, the cosine similarity metric. He also introduces Euclidean L2-distance, which measures the length of a straight line between two points in space, and Euclidean L1-distance, which measures the sum of magnitude of the vectors on each coordinate axis (i.e., Manhattan or taxicab distance).

Video Transcript

A vector space is a fairly rigid structure of numeric vectors with matching dimensions and a limited set of operations: multiplication by a scalar, vector addition, and subtraction. However, it offers a wide spectrum of similarity metrics. The simplest is dot product or sumproduct, which is a sum of element-wise products and can be any real number. Generally, the closer the points are in space, the larger is their dot product. It is cheap to compute but difficult to interpret.

On the other hand, a cosine similarity scales each vector to length 1. Or we call this length in magnitude or a norm as well and then applies the dot product. This yields a value close to 1 for vectors in the same direction and close to negative 1 for vectors in the opposite direction. Or you can think of those as points opposite on other side of 0 or the center of the Cartesian system of coordinates. Vectors which are located on the same line have an angle of either 0 or 180 degrees and hence result in cosine similarity of exactly 1 or exactly minus 1. Perpendicular or orthogonal vectors with 90-degree angle in between have a cosine similarity of 0. This measure is slower to compute but is easier to interpret.

Another measure that is easy to interpret is correlation. This is just the cosine similarity for centered or de-meaned vectors. It also varies between negative 1 and 1 and has some strong statistical guarantees. However, centering is expensive and this measure is less popular in NLP.

Euclidean 2 distance or L2 distance between vectors x and y is just the length of a straight line between those two vectors or those two points and can be computed as a root of dot product of a vector x minus y with itself. Because of squaring the large values, the element values burst in size, making this metric very sensitive to outliers. Euclidean 1 distance is a sum of absolute elements, all of the different vector x minus y. This measure is less

sensitive to the effect of outlying elements. It is also called L1 or Manhattan or taxicab distance because it is a path — or measures the length of a path between two points with 90-degree turns in between. Not a straight line in general; just like a taxicab would drive around Manhattan to get from point A to point B.

Back to Table of Contents

Watch: Examples of Similarity Metrics

In this video, Professor Melnikov demonstrates a variety of ways you can use Python to compute similarity metrics, including dot product, cosine similarity, and Euclidean distance (i.e., L2 distance). He also walks through an example in which he calculates the "closest" vector to a target vector using these different metrics.

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

Libraries/Methods

`numpy` **Library**

- `@`
- `dot()`
- `inner()`
- `norm()`

`scikitlearn` **Library**

- `cosine_similarity()`

`scipy` **Library**

- `cosine()`
- `euclidean()`

Video Transcript

Let's review a few different ways we can compute vectors and distance similarities in Python. So here we have two vectors, A and B. "Dot product" between them would indicate some sort of similarity in some sense, and that's just some of the element-wise multiplications. Because these are packed as a list, we cannot apply element-wise functions, such as those in NumPy library, but we can still use some operations in Python to do the element-wise products. For that, we would wrap these two lists into a single list of tuples, where the tuples represent the pairs in the corresponding positions of the original list, A and B. After that, we would iterate through each element of the list so each tuple is retrieved and the elements of the tuple are multiplied. Finally, all these different products — 1 multiplied by 0, 1 multiplied by 1, 1 multiplied by 2 — are added up together into a number 3; that's dot product.

There are several different ways to compute that directly with the NumPy array; if you convert A and B to NumPy vectors or representations, you can then multiply A by B, and that will do the element-wise multiplication for you automatically. That could be summed up at the end. There's also this very nice notation with the @ symbol, which will do the dot

product for the vectors and matrices. Then there is np.dot and np.inner, which is little bit more general function but still does the same multiplication and summation at the end. So all this produces the same results: 3, 3, 3, 3.

The cosine similarity is basically a dot product for scaled vectors. First, we're computing the length or magnitude of vector A and vector B, then we're rescaling vector A and B — and we're not changing direction — we're just rescaling them to have unit length — and doing dot product at the end, which gives us 0.77 as the cosine similarity. Remember, this is the cosine of the angle between the two vectors A and B. It's close to 1, so we have an idea that these two vectors are fairly similar with respect to this metric. We can compute the radians and even degrees between the two vectors by first applying the arc cosine function, which is the inverse cosine, to the cosine similarity. This will cancel out the cosine effect, and we'll just return the angle in radians, which is 0.68. Then we can apply the math degrees, which will give us 39 degrees for the 0.68 radians. Alternatively, we can divide the radians by pi, multiply by 180, and get the same representation in degrees.

There's a function in scipy.spatial.distance called cosine. This is a distance function, not the similarity. If we subtract it from 1, we will get the similarity of 0.77 just like before. There is a very nice function that will compute pairwise cosine similarities for a set of vectors. It will give us a matrix where each row represents one vector, XYZ, and the column represents XYZ as well, so the number in this matrix is just the cosine similarity for the row and column representation. On diagonal, we have all 1s always, and that's because the vector cosine similar with itself has 0 degrees — 0 in radians or 0 in degrees angle, and cosine of 0 is always 1. So that's why we have 1s. The 0s imply orthogonality or orthogonal vectors, and they are orthogonal by design, so we picked vectors that reside on the axis; [1, 0, 0] and [0, 1, 0], and [0, 0, 1], they all lie on the axis. And they can be of any length, and that would still produce a 90-degree angle between them. We can see that those angles are 90 degrees if we convert this matrix of cosine angles to just angles.

Now, we can move on to the distance function, and having two vectors, A and B — [0, 1] and [0, 2] — We can compute the distance between them. Again, they lie or reside on the y-axis, and the distance between is just 1, which is exactly what we can compute with four different ways, using norm function or Euclidean function, or just computing the Euclidean distance ourselves by subtracting the vectors, multiplying their elements or squaring their elements, adding them up, and square-rooting at the end.

Here's a nice visual representation of several different vectors that we pack into a dictionary called vecs. Vectors b, c, d, e, and f are all spread out. We are looking for the

vector that is closest to a. And "closest" could have multiple meanings; it could be either distance close or cosine similarity close. From the cosine similarity perspective, we can order them by cosine similarity. So by iterating through this dictionary, pulling every vector representation and its name, we can find the cosine similarity between that vector's coordinates and the coordinates of the vector a of interest to us, and then taking the results that are rounded and sorting them by a decrease in cosine similarity. We'd have vector b as the closest to a, and this makes sense. If we look at the picture, the angle between b and a is the smallest. Doing the same thing, same type of shuffling, and applying a different function, Euclidean distance would get us the closest vector f in this two-dimensional space. That also makes sense because the distance between a and f is the closest among all the other distances between a and any other vector.

Back to Table of Contents

Tool: Similarity Metrics

In the upcoming coding exercises in this course, you will investigate and compare three key metrics used in NLP and machine learning. Use this tool to review methods for computing each metric in Python as well as the pros and cons of using each metric to evaluate similarity.

Please complete this activity in the course.

Back to Table of Contents

Code: Generate Similarity Metrics

In the previous video, Professor Melnikov demonstrated how to implement three key metrics — dot product, cosine similarity, and Euclidean (or L2) distance — with Python. 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 use some of these techniques to calculate and visualize vector similarity.

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: Finding Similar Documents With TF-IDF

Now that you've explored the concepts of similarity and dissimilarity in more detail, you can begin to apply these metrics to gain a better understanding of similarity at the document scale. Document similarity can be calculated on the documents' numeric vectors drawn from rows of the TF-IDF matrix. You can use dot product, cosine similarity, and other similarity measures.

In this video, Professor Melnikov demonstrates this by using cosine similarity to identify documents that have similar words.

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

Video Transcript

Recall that a document-term matrix or TF-IDF matrix is a set of sparse row vectors representing documents. For any pair of such document vectors, we can compute their cosine similarity. Thus, a given document vector can give us the similarity to another document in the set of documents in that matrix. We can also find a pair of two most similar documents in our set. A major drawback in this process is that we must recompute TF-IDF matrix every time we need a vector or some sort of analysis for a new document; we'd have to add that document into the corpus and recompute TF-IDF.

Let's look at this in the code. Here we have documents or sentences that are quotes from famous people around the world about the language. We can create TF-IDF matrix, which is a sparse matrix, and then represent it as a dense matrix by wrapping it into a DataFrame. We have all these different sentences and we have all these different words that are cleaned up a little bit lowercase then with stopwords removed. Some of these have similar roots or lemmas, and our goal here is to find sentences that are most similar to the first sentence, "A different language is a different vision of life." If we look at the coefficients, some of the coefficients do not have any correlation with any other value for a word like "different," but some other ones' language are probably similar — make this first sentence similar to many other sentences. Yet there is one word, "life," that makes it similar to another document containing the word "life" as well. So "One language sets you in a corridor for life. Two languages open every door along the way." These two must be similar.

How can we compute this similarity automatically? How can we find the second sentence automatically? We can compute the cosine similarities using the cosine similarity function in Scikit-Learn, and this will pairwise create similarities for every vector in our original DTM or TF-IDF matrix. This particular number here, 0.033, is computed by taking the first row vector and the second row vector and computing the cosine similarity between them. 0.15 is done the same way between the first sentence and the third sentence that is represented by a column. This is how the matrix is filled up and it is symmetric. So 0.39 here makes two sentences very similar to each other. These are these longer sentences. And the one on the diagonal means that the sentence is perfectly similar to itself.

If we look at the row of this matrix — which is a square matrix, of course, 15 by 15; that's how many sentences or quotes we have here — the biggest value of this matrix in the first row is between the first sentence and the third one, and that's the one that we found earlier to be most similar to each other. All the other ones have smaller values. We can also identify the clusters of documents that are most similar like this vector, the similarity of the vectors representing these two sentences.

Notice that if we have a query document or a phrase which was never in the original set before, then we'll have problems because we don't have a vector for it. In order for us to have a TF-IDF vector for this phrase "language, nature and birds," we would have to edit to the set, recompute TF-IDF, and then find the similarity between this sentence or a document and every other document in the corpus.

Back to Table of Contents

Code: Find Similar Documents With TF-IDF

In the previous video, Professor Melnikov demonstrated how you can calculate document similarity based on a TF-IDF 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 using the model to make some interpretations about document similarity.

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: FastText Model Formats and Transfer Learning

As you've discovered, the FastText model revolutionized word embeddings because it can provide a vector for any word that had a subword in the training vocabulary. Follow along as Professor Melnikov discusses how trained versions of FastText are stored and how transfer learning of these pre-trained models can expand vocabulary and refine vectors.

Video Transcript

The introduction of FastText by Tomas Mikolov and Facebook in 2016 was a major breakthrough in use of NLP because it maps virtually any word to some vector, as long as at least one subword was present in the training vocabulary. The training model is distributed in over 150 languages and in two file formats. The smaller .vec text file is about 300 to 600 megabytes trained model. It offers pretty much the same functionality, albeit limited, as Word2Vec does. A larger .bin file is roughly three to eight gigabytes and contains a binary representation of the full pre-trained model. The file size depends greatly on the training vocabulary size; vector size, which can typically be between 100 and 1,000 scalar coefficients or model parameters; and numeric precision of these parameters.

We always prefer the larger model if production computers have sufficient memory to store and process it. This binary model handles out-of-vocabulary words, which is highly desirable. These files are trained on different languages and different corpora, so you'll need to pick the model that is most relevant to your domain. It is also important to note that you can post-train binary models on your highly specific corpus, and Gensim package makes it easy. This is called transfer learning, when we fine-tune the vectors and expand the model trained on generic corpus to our specific domain. For example, we can take a model pre-trained on Wikipedia corpus and Google News corpus, and post-train it on medical records for breast cancer patients where highly specialized terminology and abbreviations are used.

Back to Table of Contents

Watch: Building Sentence Vectors With FastText

You can represent a sentence or even a whole document with a vector, in the same way you would represent a word with a vector. In this video, Professor Melnikov demonstrates two techniques for building sentence vectors using FastText.

Libraries/Methods

`scikitlearn` **Library**

- `normalize()`

`pandas` **Library**

- `DataFrame()`
- `.sort_values()`

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

Video Transcript

There are many ways to convert a full document to a FastText vector, not just a single-word vector. A simple technique is to feed the whole string to FastText's library, which will parse it into subwords, map them to vectors, and return an equally weighted average vector representing a full sentence as a bag of words. This may generate too many irrelevant subwords continuing punctuation and spaces and weaken the signal in the vector. Instead, we can speed it up by first tokenizing the document into words and then retrieving FastText vectors for each word individually. We would still need to compute the average vector, but we can compute and assign weights ourselves in any desired way. For example, weights could be provided by TF-IDF with rare-word vectors weighted more heavily.

Let's go to the code. We are loading a pre-trained library from the internet public domain; wiki.simple is probably the simplest FastText pre-trained library you can find, and the zip file contains both the .vec and .bin, and the second one or the later is the one we'll use. "Wiki" means that it was trained on Wikipedia. This is about 2.5-gigabyte file, and it takes about 1.5 minutes to download it from the internet. It takes next 25 seconds to load it into Gensim's FastText object. We are loading wiki.simple.bin, which is already the expanded file, and specifying the Unicode encoding UTF-8, because some of the words in the model are unicoded. The vector size can be viewed with the function and it's 300 — just verifying

— and this tells us about the complexity of the words and how much resources this will utilize.

Next, we're loading some libraries that we'll be using, one of which is a function cosine distance. To turn it into cosine similarity, we're subtracting it from 1 and wrapping it up into a function or lambda function. The next cell takes a query, "Learn a new language and get a new soul." This is the query we'll be working with and comparing to itself, embedded in two different formats. One format is to parse it into words, as we're doing in this function GetSentVec. We are taking each word, passing it to FastText library, which returns a vector, except for the words that are in the stopword list if that's provided. Then we're finding the mean of all these different vectors. We might as well just find a sum because cosine similarity will standardize all vectors to unit length anyway.

The second approach is to use FastText's library directly to pass the full sentence to that library and get a vector. We will normalize both of these vectors so that visually we can compare the coefficients and their magnitudes and directions. Let's run the file again. Here are the two vectors side by side, and we can see the coefficients, for the most part, have the same magnitude and the same direction. But the cosine similarities are rather low; 0.66, we'd expect much higher, especially because this is the same sentence, just embedded slightly with different approaches. One uses all subwords for the full string and one uses words then their subword presentations, and all that is aggregated back to the sentence-level vector.

This is a graphical representation with scalars. What we're looking for in this 300-dimensional representation of these two sentences is that, again, the magnitude and the direction represented by strength of the color and the color itself is similar; observe the blues tend to match up and the reds or orange colors tend to match up as well in the same positions for both of these rows. Then we're taking all these different quotes that are now packaged as a dictionary, where the key is the name of the person and the quote is the value. We're taking all the different values — here they are — and embedding each one of those values with a sentence vector. We're using the GetSendVec function we wrote earlier to embed each one and compute the cosine similarity between the sentence vector of the quote to the sentence vector of the query that we had above. All that is packaged as a list of tuples and wrapped into a DataFrame to present nicely. We're also sorting the values or the rows by the cosine similarity. So if the closest quote we can find is the original quote itself, naturally it has cosine similarity 1; "Learn a new language and get a new soul" is the precise quote we had. Then the second one has 0.84 probably because

it has the word "soul" and the word "language" in it and so on. Now we can see which ones are more alike or not. Keep in mind that this treats every sentence as a bag of words regardless of the position in the sentence and relevancy to other words.

Back to Table of Contents

Code: Build Sentence Vectors With FastText

In the previous video, Professor Melnikov introduced two techniques for building sentence vectors with FastText and using them to investigate document similarity. 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 create your own FastText model on a subset of a dataset to answer a specific type of question.

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: Finding Similar Documents With FastText

Once you've calculated the similarity of the sentence or document vectors, you can use those similarity metrics to identify which documents are most similar to a given document. In this video, Professor Melnikov demonstrates the steps you can take to use similarity metrics to compare the document vectors and ultimately find similar documents.

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

Video Transcript

We might actually be ready for real-world application of NLP in document search. First, we convert each document to a 300-dimensional vector, then compute and evaluate a cosine similarity and matrix among all pairs of documents to identify most-related documents. Finally, we will convert a query string and look for a document closest to it.

So let's look at the code. We are bringing a zip file with a pre-trained FastText model from the public domain and that's about 2.6-gigabyte file. It takes about a minute and a half, and then we're loading it into a Gensim FastText object. We'll specifically load in the binary model because that will handle out-of-vocabulary words, indicating the encoding to be UTF-8 — the Unicode encoding because some of the words there are Unicode, using Unicode characters.

Then we use three different versions of the word "language" and the word "life"; We want to see which ones are similar and how similar they are. We generate a vector for each one of these words. Notice that each one brings a vector back regardless of capitalization, regardless of the plural or different form or morphology of these words. The first three words relating to "language" appear to be similar; they have similar magnitude and direction of the coefficients. If we're looking at the zero dimension, the magnitude is very large positive for the first three and "life" has close to zero. Then the next dimension has very large negative for the first three words and close to zero for the word "life." This can also be observed in color with this matrix. The colors will be similar for the first three and different for the last one. Will not always be the case, but for the most part we'll see three blues and a red or three reds and a blue or a light red.

Now, we can load this dictionary of keys as the individual names and the values as the quotes that they produce — or generate — or ideate. We're interested in observing the same coefficients, except we don't want to look at these coefficients individually; let's look at them in aggregated way. This is 15 rows for 15 different quotes — we don't see all the quotes here — and 300 dimensions for the dimensions of the FastText vectors. We will see that some of the dimensions are similar in color, and that's because they all deal with language, so wherever you see that, that's probably the aspect of language that is being captured. Some of them are different and some of these words would probably result in greater similarity to some other ones; not words but sentences. This matrix is dimensions 15 by 300 as we expected.

Now let's generate a pairwise cosine similarity using cosine similarity from SciKit-Learn. We're wrapping the cosine similarity into a DataFrame, giving it index names and column names as the quotes that we use. It's a little bit easier to identify which numbers are generated — or which sentences generate which numbers. The way you read this symmetric matrix is it's full of cosine similarities of the intersections of two words — or two sentences; one is in a row and one is in a column. This 0.73 is the cosine similarity between "Language is to the mind more than light is to the eye" in this sentence here, which is difficult to read because it's vertical. But we have 1s on the diagonal, and that's because the sentences are exactly similar to themselves so the cosine similarity is 1. Then we have blues where the cosine similarity is low. So "A mistake is to commit a misunderstanding" is a sentence that appears to be very low correlated to all the other sentences, except maybe this sentence here. And there is this 0.91 of two sentences that are strongly similar to each other for some reason. We're looking at semantic representations of the words. Notice that we're not taking the order of the words into account; there are other models that we'll look at in the course that will do that.

Finally, we want to do a search. This search will allow us to use any type of phrase and return ordered sentences by their cosine similarity. So the top match will be at the top and the lowest match will be at the bottom. So "gardening ideas"; we know we didn't have the word "gardening" before, but we have "gardening ideas" and it brings up something about growth. We can change this to misspelled version of this — let's say "gardening ideass" — and run it, and we still have something about "grow" at the top. We can change this to "anotherSoul"; "anotherSoul" may be misspelled, maybe "Soul" was a capital word, and we are still getting a relevant phrase at the top, something unrelated to the second "soul." This is how robust this model is; it will generate a vector — a reasonably meaningful vector from anything you give it.

Code: Find Similar Documents With FastText

In the previous video, Professor Melnikov stepped through how you can use the FastText model to compare documents based on their similarity. 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 use the FastText model to answer questions based on document similarity.

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 Three — Measuring Similarity Between Vectors

In Part Three of the course project, you will build a document vector and identify words within the document that are semantically the closest to that vector using several different metrics. 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.

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 up to three hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Measure Similarity Between Vectors

To express the fuzzy similarity between document vectors, you can use several different metrics that result in a numeric similarity score. In this module, you explored similarity and distance, interpreted several similarity and distance metrics, and worked with both TF-IDF and FastText to build sentence vectors and find similar documents.

Back to Table of Contents

Thank You and Farewell

Congratulations on completing "Transforming Text Into Numeric Vectors."

I hope that you now have a better understanding of what NLP is, how it can be used in your workplace to address real problems, and the basic techniques you can use to preprocess text.

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

Sincerely,

Oleg Melnikov

Oleg Melnikov
Visiting Lecturer
Cornell Bowers Computing and Information Science
Cornell University