CIS576: Conducting Semantic and Sentiment Analysis

Table of Contents

Module Introduction: Train a Model To Predict Named Entity Tags Within a Text

- Watch: Introduction to Named Entity Recognition
- Watch: Generate Named Entity Labels
- Code: Practice Generating Named Entity Labels
- Watch: Introduction to IOB Tagging
- Watch: Apply Named Entity Recognition (NER) to Texts Using a Predictive Model
- Code: Practice Applying NER to Texts Using a Predictive Model
- Assignment: Course Project, Part Two — Training a Model to Predict Named Entity Tags Within a Text
- Module Wrap-up: Train a Model To Predict Named Entity Tags Within a Text

Module Introduction: Conduct Sentiment Analysis Using Various Techniques

- Watch: Introduction to Sentiment Analysis
- Watch: Sentiment Analysis Metrics
- Watch: Introduction to Lexicon-Based Analysis
- Quiz: Sentiment Analysis Terminology
- Watch: Conduct Sentiment Analysis with the VADER Model
- Code: Practice Conducting Sentiment Analysis with the VADER Model
- Watch: Conduct Sentiment Analysis With the TextBlob Model
- Code: Practice Conducting Sentiment Analysis With the TextBlob Model
- Assignment: Course Project, Part Three — Conducting Sentiment Analysis Using Various Techniques
- Module Wrap-up: Conduct Sentiment Analysis Using Various Techniques

Welcome to Your Course

Video Transcript

NLP's popularity may have skyrocketed in recent years, but it has been around for a quite long time. Lots of scientists from many different disciplines have made numerous contributions to this field. Before deep learning came about, classical NLP enjoyed creative discovery of a surprisingly effective family of algorithms and annotated datasets. We will look at some of these datasets in this course. One example is WordNet model, which is a database of the most common words in English and other languages. These indexed words were carefully studied and labeled by community members, resulting in successful applications that are still valuable today.

In this course, we will investigate WordNet and other models that rely on carefully curated rule-based algorithms to accomplish tasks in identifying named entities — which are proper names in a document — and document sentiment — which is a subjective opinion score, such as positive or negative attitude. We will complete this course by training a deep neural network classifier to categorize movies as positive or negative. Let's get started.

Course Description

We have all been misunderstood when sending a text message or email, as tone often does not translate well in digital communication. Similarly, computers can have a hard time discerning the meaning of words if they are being used sarcastically, such as when we say “great weather, huh?” when it’s raining. If you are automatically processing reviews of your product, a negative review will have many of the same key words as a positive one, so you will need to be able to train a model to distinguish between a good and bad review.

This is where semantic and sentiment analysis come in. In this course, you will examine many kinds of semantic relationships that words can have (such as hypernyms, hyponyms, or meronyms), which go a long way toward extracting the meaning of documents at scale. You will also implement named entity recognition to identify proper

nouns within a document and use several techniques to determine the sentiment of text: is the tone positive or negative?

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

What you'll do

- Use WordNet to perform semantic analysis and compute the similarity of texts
- Use named entity recognition to locate proper nouns within a document
- Learning objective 3
- Apply analytical techniques to a document to determine directional sentiment

Faculty Author

Oleg Melnikov, Ph.D.

Visiting Lecturer

Cornell Bowers Computing and Information Science

Cornell University

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

Read: Using Python and Jupyter Notebooks in This Course

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

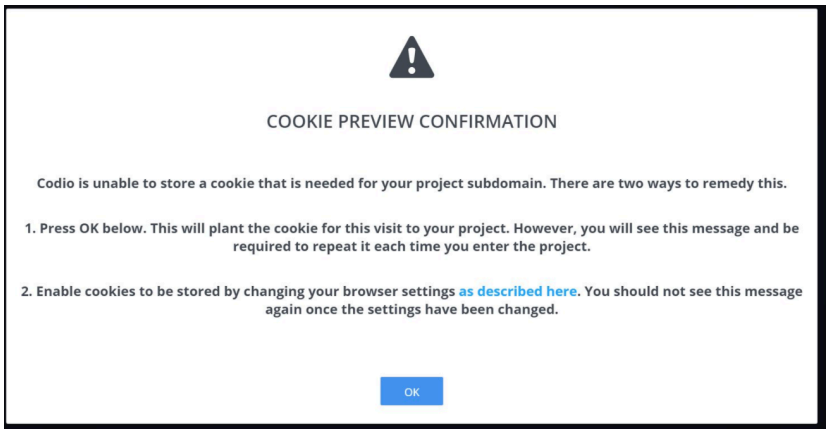
You will interact with Python throughout this course by writing and running code in workspaces hosted on Jupyter Notebooks. Our cloud-based implementation of Python and Jupyter Notebooks means that you can complete this course without installing Python on your machine.

About Your Workspace

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

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

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



Follow the instructions to resolve this issue and continue.

Jupyter Notebooks in This Course

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

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

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

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

Best Practices

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

Back to Table of Contents

Tool: Writing and Navigating Jupyter Notebooks

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

Please complete this activity in the course.

Back to Table of Contents

Read: Preview Your Course Project

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

☆ **Key Points**

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

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

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

Course Project Parts

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

– **Course Project, Part One — Conducting Semantic Analysis Using WordNet**

In this first part of the course project, you will find related sysnets to a lemma and their hypernyms. Then you will compute the path similarity score for two lemmas.

– **Course Project, Part Two — Training a Model to Predict Named Entity Tags Within a Text**

In this part of the course project, you will train a CRF model to predict NE tags to recognize movie related named entities and then measure the models performance.

– **Course Project, Part Three — Conducting Sentiment Analysis Using Various**

Techniques

For the final part of the course project, you will write functions to expand VADER'S lexicon using TextBlob's sentiment analyzer on words that are missing from VADER's vocabulary. You will then measure the f1 score for both the initial and expanded VADER lexicon and compare the results.

Back to Table of Contents

Discussion: Project Forum

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

To participate in this discussion:

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

Important Note Regarding the Sharing of Python Code

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

Back to Table of Contents

Quiz: Datasets Disclaimer

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

Please complete this activity in the course.

Back to Table of Contents

Module Introduction: Conduct Semantic Analysis Using WordNet

One way to compare the contents of many texts is by examining their vocabularies. However, even texts relating to the same topic may not include the exact same set of words — synonyms, part of speech variations, and the use of broad versus specific terms can make it challenging to compare textual vocabulary alone. The process of semantic analysis can help provide insight into the meaning, or sense, behind terms, as well as the relationship between different terms. A computer can then use these relationships to compare document content and categorize documents at scale.

In this module, you will be introduced to many types of lexical semantic relationships, which you will practice retrieving in Python using the lexical database WordNet. Then, you will use these relationships to compare various texts and evaluate their similarity.

Back to Table of Contents

Watch: Introduction to Semantic Analysis

You can perform a semantic analysis using techniques to extract, represent, and store the meaning of texts. To allow for automated methods that give more discrete semantic classifications than simply extracting meaning from collocated words, experts created WordNet, which is a lexical database containing semantic relationships and meanings for words and phrases. Here, Professor Melnikov introduces the WordNet database, describes its architecture, and discusses how it's used in semantic analysis.

Video Transcript

Semantic analysis is a broad description of techniques focused on extracting, representing, and storing meaning of text. We have already seen text embedding systems that automatically capture some semantic representations of words, phrases, sentences, documents learned from their collocations in text. This automation is highly valuable, but is difficult to control.

For example, embeddings of a word cat is continuously represented in a vector space with some degree of similarity to animal, man, or computer-assisted translation. But we might need a discrete classification of cat into this and some other specific categories.

Thankfully, experts have long been building lexical databases or lexical ontologies, such as WordNet, which contains semantic relationships and meanings which curators consider important for a set of NLP tasks. And often, such ontologies represent words and phrases as nodes of a tree graph, also known as lexical taxonomy or taxonomic trees.

Just like biological taxonomy, WordNet hierarchically classifies words but also allows storing various attributes such as parts of speech, definitions, lemmas, examples of usage, multiple senses or meanings, and much more.

The key elements of WordNet are synsets, or sets of synonymous words, which express distinct concepts. You can think of WordNet as a thesaurus or a dictionary with a much greater functionality.

WordNet covers 200 languages, but its English version has 150,000 words organized in 176,000 synsets Although this database is only about 15mb in size, it has been manually curated for decades. In fact, WordNet was developed by George Miller at Princeton University in 1985 and is still being actively used and updated today.

For example, a phone manufacturing company can add its products and even services to WordNet ontology database along with descriptions of phones, components, types of problems discovered, and various other attributes. This becomes a searchable knowledge base for consumers and company personnel. In this module, we learn about WordNet database, its architecture, and its use in semantic analysis.

Back to Table of Contents

Watch: Retrieve Synsets and Lemmas Using WordNet

Synsets, which are sets of synonymous *lemmas**, are an important part of semantic analysis in WordNet. In this video, Professor Melnikov demonstrates how to access, interpret, and list attributes for synsets and lemmas in WordNet both in English and using the multilingual feature in Python.

***Note:** The word "lemma," which you were introduced to earlier in the course series, is used slightly differently in WordNet. Previously, you used lemma to mean the shared word that several words are derived from; so *dancer, danced,* and *dancing,* would all be from the lemma *dance*. In WordNet, each synonym of the word *dance* would be considered a lemma.

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

Video Transcript

In WordNet, a synset is a set of synonyms which are semantically equivalent in some contexts. In WordNet's hierarchical structure, a word is a link to a list of synsets with different meanings or senses. Each such synset has a definition, examples, and many other attributes, including multiple lemmas, which recursively lead to other words and synsets. A "lemma" is a canonical form of a word with a single meaning while a "synset" is a set of synonymous lemmas.

Let's look at synsets, lemmas, and their attributes in code. Here, we are loading a WordNet database from NLTK library and we are displaying some of the words that are in that library. There are 150,000 words which can be pulled with the method "words." We can also draw synsets for a specific word. This will be not just one synset but a number of them, because one word "dog" can have multiple meanings. Some of these meanings are in the category NOUN, some of them in the category VERB. Each of these synsets has a form of a word and then the parts of speech separated by a period and then the version numbers. There are eight of these for the word "dog."

We can pull just the verbs, if we want to, by specifying the part of speech when we're pulling the synsets. It still is returned as a list, so we would need to extract this element of the list and object synset if we want to do further manipulations with it. Here, we are

extracting just the first sense or meaning for the word "dog noun." We're not doing it with the first element of synset, we're actually specifying the synset name in the second line. And that will bring the object, not the list, but the object that we can manipulate on.

We are retrieving some of the popular attributes for this synset. In particular, name is here, part of speech, lexical name — which is more precise part of speech — definition, examples of use, lemmas. All of these lemmas are synonyms for the word "dog animal noun." And we can actually pull lemma names. In the next cell, we are pulling all the available attributes for a synset. This is a very extensive list. Some of them are leading to path similarity, and we'll study some of this later in this module.

We can also retrieve all synsets and even all synsets that are just nouns and samples from that list as well. There are 117,000 synsets and 82,000 of them are nouns. This could be explored further. You can build very complicated trees and networks of these synset structures. If you want to pull and work with a specific lemma object, then you would pull a lemma specifying this synset and then the period in the lemma name. It has four components. For that lemma, you can pull its synset, or the synset of that lemma is in. Remember that synset contains multiple lemmas. All of them are synonyms. All of them have exactly the same meaning. The lemmas have slightly different attributes, but many attributes overlap with those of synsets.

And if we want to see the differences between sets of attributes, we notice that antonyms is a method that is available for lemmas, but not for synsets. We'll see how to use this later. We can also do a bit of correction on the words that are morphologically different using the morphy method of WordNet library. So here, the word "running" is not in WordNet, but "run" is. Morphy detects this and pulls up a synset for the word "run." In this particular case, it actually pulls the word and then you can use the word to pull up a set of synsets. "Corpora," and many other plural forms, can be morphed to a singular form, but not all of the words or phrases can be found on morphy.

Synsets can also be represented in multiple languages. Here, we have Open Multilingual WordNet. This is the set of languages that are supported by WordNet. There are only a few. Russian is not here. Some other languages — I can't seem to find Chinese here as well. But Japanese is here. And what we can do is pull the word "dog" and its lemma in different languages. These are all lemmas for the word "dog" or for the synset "dog noun animal." And they're all in Japanese.

Back to Table of Contents

Code: Practice Retrieving Synsets and Lemmas Using WordNet

Previously, Professor Melnikov demonstrated how to retrieve and how to extract information on synsets and lemmas in WordNet. 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 WordNet to retrieve and find information regarding the part of speech for synsets and lemmas.

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: Retrieve Lexical Semantic Relationships Using WordNet

There are many different lexical semantic relationships that can be shared between two or more words. In this series of videos, Professor Melnikov introduces and demonstrates some of these relationships between words or synsets.

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

Entailments, Homonyms, and Antonyms

Here, Professor Melnikov defines entailments, homonyms, homographs, homophones, and antonyms before demonstrating examples in code.

Video Transcript

In WordNet, we can analyze semantic relationships and more in synsets. For example, we can retrieve or evaluate entailments, which are verbs causally evolved from some other verb. For instance, the word "eat" causes the entailment "chew," but not the other way around. This allows for constructing or assessing a sequence of logical actions. We can also evaluate homonyms and homographs and homophones. Homographs are spelled identically but have different senses, such as the words "bat" and "bat," spelled exactly the same, but they might mean "club for hitting a ball" and "nocturnal flying animal" — two different senses. Homophones sound the same, but also have different senses. For example, in "write on paper" and "turn right," the words "write" and "right" are homophones. These words are often the cause of spelling errors and problems in speech synthesis. Both homographs and homophones are homonyms, which is their superset. We can also evaluate synonyms that we've seen earlier and antonyms, and other relations that we will see in later videos.

For now, let's look at the code and some of these examples. We are loading the WordNet database as usual. In the next cell, we are going over several different verbs and printing out entailments for each one of their synsets. We're taking just the first synset where we need to have — it needs to be a verb because the entailments only makes sense for a verb and entailments are themselves synset verbs. So the word "walk" causes entailment "step," but not the other way around. "Eat" causes "chew" and "swallow." "Digest" is similar to "eat," but it has different senses that has an entailment "consume." "Watch" causes "look," "look" causes "see," and "see" doesn't have an entailment.

Likewise, we can pull out the homographs for a particular word. We're using the word "bank," which has many different meanings even though it's written the same. Some of these are financial institutions, some of these are riverbanks along the riversides. We are excluding the words, or the synsets that are not homographs which are also related to "bank" but spelled differently, so that those would not be homographs.

In this next example, we're using the word "good," pulling its synset, pulling its first or zeroth lemma. For that lemma, we're pulling an antonym. Notice that antonyms is a method on lemmas, not on synsets, which can be confusing. But for the word "good" and its first lemma, which means "feeling good," "bad" would be the antonym. We can likewise pull some of the derivatives of the word "good," which is an adjective, first version synset, and "goodness" would be one of these derivative forms.

Hyponyms and Hypernyms

In this video, Professor Melnikov introduces the parent-child relationship between hyponyms and hypernyms of a synset. In code, he shows how to list hyponyms and hypernyms for a synset and how to determine the similarity between two synsets by calculating the path distance between them.

Video Transcript

A sense is a hyponym of another sense if the former is more specific than the latter, which is called a hypernym. For example, a vehicle is hypernym of a car which is a hyponym of a vehicle in this parent-child relationship. This induces another tree structure on words, and we can use it to determine the path-like distance between two words in this tree. Hence, we compute similarity and dissimilarity for a pair of words in WordNet database.

Let's see some of this in code. After loading the WordNet database, we are retrieving a synset for the word "dog," "domestic animal" in particular, and pulling all its hyponyms to be displayed here on screen. There are 18 of those. Some of those are breeds, some of those are different types; "puppy" is here as well, "toy dog," and so on. We can likewise pull all the hypernyms of the word "dog"; because it's a treelike structure, there should be fewer, but not always. So "canine" and "domestic animal" are parents of the word "dog" and they branch out and lead to the word "dog" eventually, to the synset.

We can compute the distance in this tree between any two synsets, like the "dog domesticated animal" and "domesticated animal cat" have a distance of 0.2. The "dog" and "puppy," both being the animals, have a higher similarity. But if we change "dog" to verb, which is "to chasten," then the similarity drops. We can also present all different paths from the top of the tree, which is entity synset, to the particular synset. In this case, "puppy noun version one," or the dog's puppy. Now, all these paths are exactly the same up until we get to "animal." After "animal," the path branches out into different types of animals before it's recombined into "puppy." This is a recombining tree. It still has a root that is an entity synset, but before it gets to "puppy," some of the paths are longer, some of the paths are shorter, as we can see here.

Here's another example with the word "tiger." This definition of "tiger" is for "an audacious person." And there are two different paths starting from the entity and leading up to the same "tiger" definition.

Holonyms and Meronyms

A given word can have holonyms, which is something that the initial word is a member of, and meronyms, which are parts of the initial word. Professor Melnikov defines these terms and illustrates examples of holonyms and meronyms for words in code.

Video Transcript

Often, we would evaluate whether words respect part-whole relation. In this case, the whole is a holonym and the part is a meronym. For example, lettuce and tomato are meronyms of a sandwich, which is their holonym. Let's see a few examples of different types of this relation in code. We are loading the WordNet database and building a function that will pull different types of holonyms and different types of meronyms. There are holonyms that represent membership of a particular concept or part of a particular concept or a substance.

Let's look at what "tree" synset gives us with respect to these definitions. So a tree is a member of forest, but its parts are burl, crown, stump, trunk, and so on. The substance of the tree, that's if you cut the tree and look at the cross-section of its inner circles, you will see heartwood and sapwood, which is more like a substance category.

Similarly, we can look at holonyms and meronyms of "water." Not everything is filled up here as far as the holonyms and meronyms, but water consists of hydrogen and oxygen and makes up a body of water, it makes up ice, ice crystals, perspiration, and so on. The database isn't perfect, so not everything is going to be defined fully, but it can be extended.

An "atom," on the other hand, is a part of chemical element or a molecule, and it contains some elementary particles, a nucleus in particular. We can also look at "DNA" or the DNA sequence and what it contains. The meronyms or the parts of the DNA are base pairs and genes, nucleic acids, but substance-wise, those are ACGT nucleotides — adenine, cytosine, guanine, and thymine.

Similarly, "fish" can be looked at as a part of school, a school of fish, rather, and also has components such as fin, fish scale, fishbone, and so on. And "ear" is part of the face, or the head, rather, and auditory system, but it has meronyms such as auricular artery, eardrum, and some other components.

Another example is "kitchen." Kitchen is part of the dwelling. There are lots of components in the kitchen, and kitchen can be part of many different systems, but it isn't filled in completely.

Back to Table of Contents

Code: Practice Retrieving Lexical Semantic Relationships Using WordNet

Previously, Professor Melnikov demonstrated how you can retrieve and analyze different lexical semantic relationships in WordNet. 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 WordNet to analyze entailment, antonyms, and hyponyms, and how to compute path similarity between a word and two homographs.

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: Use Cases for Lexical Semantic Relationships

In this module, you've reviewed WordNet's synsets and lemmas, which allow you to group words on the basis of similarity. They also enable you to identify a specific sense, or meaning, associated with a single word. You've also used other terms to describe various relationships between words, including parent/child and part/whole relationships.

Use this tool to review the types of lexical semantic relationships examined throughout this module, as well as how to access them using WordNet.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Semantic Analysis Terminology

In this module, Professor Melnikov introduced several lexical semantic relationships that are useful for performing semantic analysis. Test your understanding of these terms and relationships in the following quiz. Note that in some cases, a question may have more than one correct answer. In those cases, select all answers that apply.

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: Compute the Similarity of Lexical Semantic Relationships Using WordNet

You can determine basic document similarity by computing the semantic similarity between two synsets in WordNet. Watch to see Professor Melnikov give an example in code using a set of synsets to compute the similarity score and find the closest hypernym between each pair of synsets. He also shows how you cannot always compute similarity between two synsets that are different parts of speech and explains why that is important to consider when thinking about document similarity.

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

Video Transcript

The WordNet taxonomy is used to compute semantic similarity between two synsets of the same part of speech. This could be two nouns or two verbs. These are basic building blocks for similarities between sentences and documents, which are essential in document search as well as clustering, classification, and many other applications.

Let's construct document similarity in code. We are loading some of the libraries that we'll need from NLTK in the first cell, and in the second, there is a TsWords tuple that has words for which we are pooling all the first or primary synsets. We don't care about the others for now. Just the first ones will turn out to be all nouns, so we have n.01 for each one of them. Notice that the first synset or sense for "tiger" is not an animal but "a fierce person." This will be important when we start relating or building paths between all these different synsets. The others are either animals or objects or organisms, and two persons; one is Ronald Reagan, the president of U.S., and Bill Gates, the CEO and president of Microsoft.

In the next cell, we're building up a matrix of closest hypernym and the path score. The similarity score can be thought of as a percentage; so 0.25 similarity between "cat" and "lion" is also 25% similarity between these two synsets — well, these are words, but there are synsets behind them that represent them that we pulled out from taxonomy of WordNet. So similarity between "cat" and "lion" is relatively higher than similarity between "cat" and "tiger," and that's because "tiger" is a person, not an animal. So they are related through the hypernym "organism" — that's the closest hypernym we can find in the tree. Then "Ronald Reagan" and "Bill Gates" are related through hypernym "person" with a 10%

similarity. All the diagonal elements, of course, have the same hypernym as the words coming into the path function or path calculating function. So "cat" and "cat" have hypernym "cat" with 100% similarity.

This is not sufficient to build a good document similarity because oftentimes we will be comparing, in the algorithms that we'll display in a moment, we'll be comparing different parts of speech. Let's see what parts of speech are not compatible with each other. Nouns are located in the same tree, so we can find similarity between two nouns as long as they are found in WordNet. But nouns and verbs are not always path connected because they live in different trees. Verb trees are different from noun trees, so "cat" and "run" do not necessarily have a path. It is sometimes possible to find the path between verbs and adjectives, so "run" and "red" or "run" and "slowly" adverb here also have connection of 20%, whatever that means. So we need to be mindful when we're taking two different documents, parsing them into words, into related synsets, and then trying to find paths between those synsets because those paths will not always exist.

Back to Table of Contents

Code: Practice Computing Similarity of Lexical Semantic Relationships Using WordNet

Previously, Professor Melnikov demonstrated how to compute the similarity score and find the closest hypernym between a pair of synsets. 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 evaluating the similarity between different senses of the same word.

All practice exercises are optional and ungraded.

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

Please complete this activity in the course.

Back to Table of Contents

Watch: Compute Document Similarity Using WordNet

An important initial step to computing document similarity with WordNet is to tokenize and determine part of speech (POS) tags for each word. You can then find which synests are closest to representing words with the same POS, and you can calculate path similarity. From there, you can find the average similarity between two lists of synsets or words. Here, Professor Melnikov demonstrates how to compute document similarity using these methods in code.

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

Video Transcript

Now we're continuing to build up the document similarity from what we've learned about similar synsets. Here we have the dictionary which converts the Penn, or Penn State, POS — parts of speech — to WordNet POS. WordNet POS are N-A-R-V, there are only four of them. And in SplitNTag function, we are taking the document as a string and a default POS tag. Some filters like the list of stopwords and minimum length of the word. We are tokenizing that sentence or document and computing the POS tags. These are still NLTK tags or Penn State tags, POS tags, which will be replaced and the words will be filtered out. As a result, for the "I ate a red bean.", we have on the "ate," "red," and "bean" was their respective POS tags. POS tagging is not perfect. Even humans cannot perfectly identify it, so there will be some noise and errors, mistakes when it's done automatically.

In the next function, we are determining the synsets which are closest to representing these words and their POS. In "I rode the bike," "ride" and "motorcycle" — verb and noun — are returned as a list. And in "brand new laptop," only "trade_name" for "brand" — and it's a noun — and "new" for "new" — it's an adjective — are returned. The "laptop" is identified as an adjective. And there is no "laptop" adjective in the taxonomy of WordNet, so it returns none and it is skipped.

In the next cell we have a function SynSim, which uses path similarity to compute the distance. The only reason we need a function wrapper for this is to treat non-cases. There will be occasional non-cases between nouns and verbs, for instance; we'll replace those

with zero similarity when those are found. Next we have SynsSim, very similar function to the one before, except it takes two lists of synsets and it determines the closest synset in the second list for every synset in the first place. This is specific to the list order, so we will call this function twice so that we can compute the synset similarities regardless of the order.

Notice that when we run this function with debug turned on, which allows us to view the computations in between, the intermediate results are returned. So list of synsets one and two in this specific order will give us the pairs "lion" and "tiger" with 0.33 similarity and so on. There will be four of these pairs because the first list has four synsets. In the different order of arguments, the first synset list has two synsets and we only have two pairs returned. In either one of those cases, the average similarity between the synsets is returned, so 0.33 plus 0.33 divided by two is still 0.33. If debug is turned off, none of the intermediate results are returned, and we're only returning the final average similarity between two lists of synsets. This function is wrapped, again, to average the similarity between two lists regardless of the order; we're taking the average of those ordered arguments.

Finally, we have these several sentence pairs where the similarities are computed. "It rains outside" and "rain is outside" have "rain" as a verb and the noun, so there will not be a path between them. "Outside" in the first sentence is a noun; in the second sentence it's an adjective, so there will not be a path either. "Is" and "rains" are verbs, so there is a path between them. It's only 10% similarity between the first two sentences, but the second one has a higher similarity of 30%, and that's because "pouring" and "rains" are verbs located in the same taxonomical tree with the high similarity between them. Same goes for the other sentences. "Brand new laptop" has low similarity as expected, but somewhat similar to the first pair. Be mindful of these pitfalls because they do come up as you are trying to compute document similarities.

Back to Table of Contents

Code: Practice Computing Document Similarity Using WordNet

Previously, Professor Melnikov demonstrated how to compute the average similarity between two lists of synsets in order to find document similarity in WordNet. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice computing the similarity score to find document similarity between multiple documents.

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 One — Conducting Semantic Analysis Using WordNet

In Part One of the course project, you will find related sysnets to a lemma and their hypernyms. Then you will compute the path similarity score for two lemmas. The tasks you will perform are based on videos and coding activities in the current module but may also rely on your preparation in Python and basic math.

Resources

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

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

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

Completion of this project is a course requirement.

Instructions

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

Please complete this activity in the course.

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

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Conduct Semantic Analysis Using WordNet

In the previous module, you used semantic analysis to examine the relationships between words and evaluate the similarity between documents. First, you defined several lexical semantic relationships and practiced retrieving them with WordNet. After comparing the similarity between different words, you extended these techniques to compare the similarity between documents.

Back to Table of Contents

Module Introduction: Train a Model To Predict Named Entity Tags Within a Text

Another way to compare documents is by noting whether they reference the same person, location, or date. *Named entity recognition (NER)* is a technique for extracting and categorizing these key entities, which are often proper nouns.

This module will introduce you to a variety of named entities before you practice identifying them within a document. After training a model to predict NER tags, you will then test its performance on an unfamiliar document.

Back to Table of Contents

Watch: Introduction to Named Entity Recognition

A word that is used to represent real world objects and that hold greater semantic meaning are called named entities. You can use the named entity recognition or extraction (NER or NEE) technique to recognize named entities in a text. In recent years, this technique has been found to have a success rate comparable to humans. In this video, Professor Melnikov introduces named entities and discusses some common use cases of NER.

Video Transcript

A named entity is a word or a phrase which represents real-world objects, such as a person, an organization, a country or location, an event, and many others. It is an instance of an entity and it is often denoted with a proper noun. For example, "New York City" is an instance of a city. "Bill Gates" is an instance of a person. "Microsoft" is an instance of an organization. Named entities often encapsulate a much greater and richer semantic meaning than an instance it is derived from. So we can benefit substantially from recognizing named entities in text using a technique called named entity recognition or extraction, NER or NEE.

As a use case, consider a message from Lee to Kira, "Skype me at 4:00 p.m. today." Wouldn't it be great to automatically add this event to Kira's calendar? As another use case, consider millions of news articles that we need to scan and find all related, competing, and partnering organizations. We could sell this service to any company that wants to know the players in its industry. In yet another example, consider NER in tweets and news in order to identify trends that might affect the financial position of publicly traded companies. For this, we might also need a document sentiment analysis, which we will cover later. Then some automated trading algorithm can buy or sell positions of the trending company or make real-time recommendations to subscribers or hedge funds.

A related task is named entity linking or disambiguation, NEL or NAD. Here we assign a unique identifier to the entities. For example, in the sentence "Paris is the capital of France" we want "Paris" to be recognized as the city, not Paris Hilton the person. Also, a corpus may have named entities such as "Tesla," "Tesla Motors," "Tesla, Inc.," "Tesla Corp" that all refer to the same entity, but we prefer all of this to be linked to the same "Tesla Motors" entity. This in turn relates to coreference resolution problem, where a mention of

"it" needs to be linked to the named entity. For example, "GM's new electric vehicle will boost its '21 revenue" will have the words "it" and "GM" refer to "General Motors," which we would like to identify automatically.

Despite all of the ongoing challenges, NER has dramatically improved in recent years with the transition from rule-based training to trainable models. Now the state-of-the-art systems can achieve 93% or higher F-measure performance on some standard annotated datasets compared to human performance of about 97%.

Back to Table of Contents

Watch: Generate Named Entity Labels

You can use the spaCy software package in python to parse text and use NER to identify and tag named entities as a date, event, geographical location, etc. Watch to see Professor Melnikov demonstrate this in code for a single sentence and for a presidential inaugural speech. He also shows how you can view the tags for the named entities as a text table or as a paragraph with colors using the spaCy displaCy visualizer.

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

Video Transcript

Let's now try to apply spaCy's named entity recognizer to some example texts. So we have here spaCy loaded along with its trained core library. This is a small version library. The larger version might give you better results. The first sentence we're looking at is "Apple bought a U.K. startup, GrapeAI.io, for $1 billion on Jan 28, 2021." This is full of named entities, and as we pass this sentence, bringing to NLP object from spaCy, it parses it out, tags it, and we can extract the entities, named entities from this resulting object. For each such named entity, which we call "e", we can extract attributes such as text, where it begins, where it ends — those are character positions in the text in the original document — and "label", which is without an underscore, which is just an ID for the tag, and the "label_" with an underscore is the tag itself. So "Apple" is an organization, "U.K." is a geographical location, "GrapeAI.io" is organization, "$1 billion" is money, and date is fully captured as well. This seems to be reasonably well captured, but it's not going to be always. So there will be errors — it's not perfect.

We can try and see the other use case with the inaugural speeches. The "inaugural" corpus contains speeches of all American presidents. You can see a particular president and the year when the inauguration occurred, how many words were in an inaugural speech, how many sentences. We'll focus on Bill Clinton's 1997 inaugural speech. We can extract that, and that's just a document of sentences. There are approximately 2,500 of them. These are ordered. So Bill Clinton's 1997 has 2,500 words and 112 sentences. All that will be parsed by spaCy's NLP object, which will also tag all words as entities, as POS, and so on. We can actually tell it what not to tag, but here we don't do that. Finally, we will wrap it as a DataFrame and transpose it so it's presented horizontally instead of vertically.

So each phrase is — not each phrase, but phrases are now tagged with their named entities and named entity tags. "A thousand" is a cardinal number, "millions" is also a cardinal, "one" is cardinal, "ten years ago" is a date. You might question some of this, whether "a new millennium" should be just "a millennium" or "a new millennium" is actually a date. It's, again, sometimes to some degree is subjective. "This remarkable century" perhaps could be just "century." We also see some geographical locations like "America." "Earth" is a location, and so on. "Congress" is organization. "Martin Luther King's" a person, and perhaps the apostrophe s could be dropped as well.

So this could be presented with colors identified in the text directly with the spacy.displacy.render method. We indicate that it's a Jupyter Notebook, so it returns HTML, which is then rendered accordingly. So we have all the dates here are colored with blue color, the geographical locations are colored in orange, and so on. We're only displaying about 200 characters, but the full text will be displayed. This could be useful for extracting more content or context around the particular tag or collecting different tags of different types, for instance, organizations or dates and so on.

Back to Table of Contents

Code: Practice Generating Named Entity Labels

Previously, Professor Melnikov demonstrated how to use the spaCy software package in python to parse text and use NER to identify and tag named entities. 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 extracting, analyzing, and manipulating NER tags.

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: Introduction to IOB Tagging

You can use inside-outside-beginning NER, or IOB tagging, on a text to generate named entity tags and mark if each word in the text is the first word of a named entity (tagged as beginning), a subsequent word in the named entity (tagged as inside), or not a named entity (tagged as outside). Each of the named entities are additionally tagged with what type of name entity it is. Here, Professor Melnikov demonstrates IOB tagging on text from the Groningen Meaning Bank corpus.

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

Video Transcript

Tagging various textual chunks and phrases is very important in NLP. These metadata are often used in downstream NLP processing, so all inaccuracies, limitations, and benefits of the chosen tagging structure is carried forward in the analysis. For example, we might use preprocessing to distinguish important words to generate better, in some sense, parts of speech or POS tags leading to better NER or named entity recognition tags leading to better word or document vectors, leading to better Q&A, question and answering, or some other terminal NLP task. So any errors in between will simply accumulate. We want to be as precise and accurate at the very beginning and at every single step of the sequence.

Let's see how IOB and inside, outside, beginning NER tagging is applied to an annotated document in code. We are loading some of the libraries that we'll need for this task. The GitHub location has this Groningen Meaning Bank, or GMB, NER annotated dataset. We are changing some of the names of the columns to make it a little bit easier to work with. But the structure is a bit unusual. Instead of a text given to us, we have a DataFrame where the first column indicates the sentence ID, the second indicates the words in those sentences in that sequential order — so don't shuffle the rows. The POS indicates the tags for the words, and NE are the tags or named entity tags for those words as well. Most of these NE tags are going to be O or "other" or "outside" of the chunk, and some of them will be inside of the chunk and at the beginning of the chunk, as we'll see momentarily. But the file doesn't have many of the IDs — they are NAs. So we need to propagate the first number down for every word in that sentence, and then the second ID is propagated down, and so on. This is done with the fillna method in Pandas.

Once we do that, we can extract any sentence, uniquely recreate it in a textual form — here's one — and we can present this DataFrame or the sentences in it with a nice colored tagging structure. We use styling for the DataFrame, which is a very nice way of coloring different values and even index names and column names. We can highlight in different colors all the different tags. We have eight NE tags and three different locations for these tags. "B" indicates the beginning of the chunk. So if this is a named entity chunk consisting of four different words, International Atomic Energy Agency, B will indicate where it begins. All the other ones will indicate how long the chunk lasts for. Anything in between is filled up with O or "outside" tags. If we don't have any inside or O in between, we have two tags starting with B next to each other. "Vienna" "Wednesday" are two different types of chunks. None of them have more than one word, so we have B-geo for Vienna and B-tim for Wednesday. Same goes for all the other tagging in the structure. This is nicely presented in color, for the person, for the time, for the geopolitical entities, and so on.

In the next cell we are computing a distribution of all these different tags. We're not counting the I tags because they are just inside of the chunk of the name, the entity chunk. We're only counting the beginning tags. We have high-value counts or high support of chunks with the large numbers and very few with these three types of named entities. When we build the model in the next video to predict the named entity tag for a particular word in the text, this will become important because our model will have lower number of observations to learn from for these three tags. We're assuming that the actual population of tags or named entities is infinite and, of course, there are other tricks that we can use to boost the performance of these three. Our goal is to train a model that will use this dataset and its tags to learn the association between different textual attributes and the tags and then use that model to predict named entities for the tags that the model has not seen before.

Back to Table of Contents

Watch: Apply Named Entity Recognition (NER) to Texts Using a Predictive Model

In this series of videos, Professor Melnkiov describes and demonstrates how to build, train, test, and use a NER predictive model.

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

Building Features for a NER Model

You will use the conditional random fields (CRF) model, which is a powerful machine learning technique that utilizes neighboring words, to learn associations between features you build and human-generated NER tags. You can build features for each word along with its POS tag in a document along with the information from the neighboring word on each side. Here, Professor Melnikov demonstrates how to do this in Python code, and how you can package your results as a dictionary of features.

Video Transcript

Now we're ready to build our own NER tagger using conditional random fields or CRF model. This model is not in the scope of our course, but it is a powerful machine learning technique which can take advantage of neighboring structure of a sequence of words. In particular, we will build features from each word and its POS tag in a document and each of its neighbors in a three-word window. The CRF will learn associations between these features and human-labeled NER tags in GMB document. Then the given model can be used to tag some brand new document and identify NER tags in it.

Let's see this in code. First, we will need a special column that we will use for model training and testing. It will combine the word, its POS tag, and its NE tag, which is already known and human-labeled, into a single tuple. We have "Thousands," and that's a noun, and it has the outside named entity tag, using IOB tagging architecture. Every word in the sentence here is presented horizontally, is now captured or presented via this WPE tuple with three elements. Then we will create a list of these triplets by exporting the column to a different variable. And for each triplet, we will create a set of, or rather a dictionary of features. Notice that we're only using the first two elements in a WPE triplet, "Thousands" and "NNS." To create features, we cannot use the named entity for feature creation because that's exactly what we're trying to predict. However, we will use the NER tag to train a model and then test its quality.

So using any word that is passed into this featurized function and its POS tag, we are generating keys and value pairs, whether the word is a digit, whether it's uppercase, title case, what its POS tag is and the first two letters of the POS tag. If it's a word that is located before the word of concern, or the center word as we call it, then we prepend to the keys the letter "b." If it's after, then we prepend letter "a." This way we can generate a very rich set of features for the word and the surrounding words. And if you need, you can expand the word window to size five and seven and so on. We use that featurized function to generate features for a sentence or sentences presented as a list of these WPE tuples. "Yahoo fell today" is a sentence with three words in it and we need to take special care of the words at the beginning and at the end of the sentence because they either don't have the word before or the word after. So we add a particular key with value 1 BOS or EOS, indicating beginning or ending word in the sentence.

So every word in this sentence, "Yahoo fell today" with three words, will have its own set of features and then we can all package them as a list of dictionaries for each word in the

sentence, for each sentence in the document. This is what we have here. We have one particular sentence extracted and the fifth and sixth words, "through London," are being demoed here as WPE tuples, and each one of these tuples is converted to a dictionary of features. So there are two dictionaries and corresponding labels that we will first use in training and then test them.

Training and Testing a NER Model

Given the generated features for a NER tagger, you can now utilize these features to train and test a model. Follow along as Professor Melnikov demonstrates how to train, test, and optimize your model.

Video Transcript

Now that we have generated features, we can use those to train a model. The model is not in Scikit-Learn, but it is very similar to Scikit-Learn and can use many of the Scikit-Learn objects like train_test_split. In fact, we use train_test_split to split the sentences, or the features generated from those sentences, which is a list of lists of dictionaries of the tuples in WPE format. We split those into train with their labels and into test set with their labels with a 25% and a random_state.

Then we use those to train the model, which is trained just like any other model in Scikit-Learn. It has a very rich set of hyperparameters which you can investigate with the help manual for this model. You can play with different algorithms, the regularization parameters. C1 is for lasso or L1; C2 is for ridge or L2. The higher number will imply a simpler model and handles overfitting well. If you see a model is performing very well on the training set and very poorly on a test set, that's an overfitted model and you need to push the C1 or C2 or both higher, and it could go into infinity. All these parameters can be tuned with a grid search in order for us to find an optimal set of parameters for a particular training set.

Once done so, we can run the prediction on the set-aside sentences or features representing the sentences and evaluate their performance. Here we have the first sentence tags that are actual and predicted. We can see that some of the text are recovered but time, for some reason, isn't. In fact, in two places the time has not recovered. This means we would look further or deeper into time tags and named entities to identify any features that can be extracted. The alternative could be to add more training data where timing is used, that has named entities of time tags, and use that to train our model specifically with the time values improved.

Here we have the classification report, which is very similar to Scikit-Learn. This one is presented by this object CRFsuite. We can see the performance of each individual tag. F1, which is a combination of precision and recall, means that higher value is better. And 0 and 1 is the full range. We want it to be as close to 1 as possible. So geopolitical named entities are tagged better than anything else and can still be improved while the inside geopolitical tags are not predicted very well as well as the time, and geo are all low performance. We would look into these named entities to see how to predict them better. The support simply indicates how many tags were used in the training set for a particular category.

Applying the Trained NER Model

Here, Professor Melnikov shows how to apply your trained NER tagger model to a sample text, then details the successes and shortcomings of the results.

Video Transcript

Once the model has been trained and optimized to our satisfaction, it is ready to be rolled out to production where it can identify and tag named entities. Note: These specific domains require a specific NER tagger and specialized feature sets; a classifier trained on financial news is likely to perform poorly on medical records or legal documents. Also, since we build our features from characters using capitalization and digits in the strings, our taggers will not generalize well to other languages.

Let's see how we can apply the model that we just trained to a sample document and extract NER tags. We have this document of a few sentences about Facebook that we can parse out or tokenize and then produce POS tags for. That can be done with NLTK. We have the first sentence, "Three more countries have joined an 'international grand committee,'" are all tagged with the POS. These are the inputs into the featurizing functions that will generate features from this tuple. Notice that those featurizing functions do not use the third element, which before was the label we were trying to predict. Here we're assuming we do not have any labels and we are trying to generate those NER tags or labels ourselves with the trained model.

We can create the features for each one of the words in the document and for each of the sentences in that document and then use the predict method of the trained model CRF to predict NER tags for this features list of dictionaries. Here we have these labels that correspond to the words in the first sentence, words from 10–20. Then we can combine this together with the words. Now we would like to extract all the named entities. Typically that's the last task. If we simply drop the O tags, it's not quite there yet because we see "Mark" "Zuckerberg" that is a named entity chunk correctly recognized, but it's not glued or joined together. We would need to somehow drop the inside tags and combine "Mark Zuckerberg" to a single named entity tag or chunk.

We can do it in a number of ways. You can do it in a loop or, as we do it here, in a vectorized fashion. We are adding one more column. Here it's presented as a row because we are transposing the DataFrame of increasing numbers. These are unique IDs and the idea is to basically propagate the ID anytime there is an inside tag. So if the inside tag follows the beginning tag, we propagate the 0 to the next position. Again, there are multiple ways of doing it. We would, as one approach, erase the IDs wherever there are inside tags and erase the inside tags and then use the forward fill method to propagate the IDs forward. We're keeping the empty strings as they are. Now we can

group by these IDs and concatenate the words in this index and the NER tags. As we do that, we have the named entity chunks glued or joined correctly. So "Mark Zuckerberg" now is one phrase that is labeled as the person. And even though "Facebook" is not a geographical entity, it is still concatenated together with the next tag correctly. "New York Times" is a difficult one; it has three words and it is correctly glued together as a single entity.

Back to Table of Contents

Code: Practice Applying NER to Texts Using a Predictive Model

Previously, Professor Melnikov demonstrated how to use IOB tagging and how to create features for, train, test, and apply a NER 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 identifying NE tags in a new document using a pretrained model.

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 — Training a Model to Predict Named Entity Tags Within a Text

In Part Two of the course project, you will train a CRF model to predict NE tags to recognize movie related named entities and then measure the models performance. The tasks you will perform are based on videos and coding activities in the current module, but they may also rely on your preparation in Python and basic math.

Resources

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

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

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

Completion of this project is a course requirement.

Instructions

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

Please complete this activity in the course.

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

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Train a Model To Predict Named Entity Tags Within a Text

Now you have evaluated the content of texts using both lexical semantic analysis and NER. In this module, you were first introduced to several named entity tags available through the SpaCy library. With the help of the GMB Corpus, you were able to practice implementing NER and IOB tagging before building your own NER tagger. Finally, you trained, tested, and applied your tagger to predict named entity tags in an unfamiliar corpus, then evaluated its performance.

Back to Table of Contents

Module Introduction: Conduct Sentiment Analysis Using Various Techniques

Lexical semantic analysis and NER are two powerful tools for evaluating the meaning behind texts. This is especially useful when categorizing many articles, such as news stories, according to content. In some cases, however, a text's content is less important than the emotion behind it. Consider a business that is evaluating its customer product reviews—how do customers feel about a particular product?

In this module, you will evaluate the sentiment behind various texts by constructing two sentiment analysis models using the VADER and TextBlob libraries. Then you will test both models and compare their performance.

Back to Table of Contents

Watch: Introduction to Sentiment Analysis

You can use sentiment analysis to classify a subjective document or text into their polarities using a simple model that is based on rules regarding the presence of certain words. Sentiment analysis is not effective on objective texts, and more advanced models can identify states beyond a single polarity. Follow along as Professor Melnikov introduces sentiment analysis, then discuss challenges and common use cases.

Video Transcript

A sentiment is a subjective opinion, feeling, or polarity towards an event, a product, or a service. In a sentiment analysis, we hope to systematically classify documents into one of the polarities. This works best for subjective texts, such as feedback surveys, social media posts, reviews for movies, vacation rentals, businesses, personal products, services, and much more. A simple model can apply lexicon-based rules to classify polarity based on the presence of expressive words and phrases such as "like," "good," "great," "amazing" and their antonyms such as "dislike," "terrible," "bad," and so on. Note that some of this can even be ordered in an increasing or decreasing degree of sentiment, for example, "good, great, best" or "pleased, thrilled, overjoyed, ecstatic." However, this gets complicated since negation can swap the polarity of a statement. For example, "I hate long air flights" is a relatively straightforward opinion. However, "Disliking long air flights is not my thing" uses negation and inverted word order, which is difficult to capture with a rule-based model due to a multitude of variants. Other difficulties arise from sarcasm, multiple attitudes towards the same object, and so on.

Sentiment analysis is not really suitable for objective documentation such as product manuals, factual reports, financial statements, legal documents, and medical charts. Advanced sentiment analysis may also identify joy, anger, disgust, fear, surprise, sarcasm, and other emotional states beyond one-dimensional polarity. A neutral polarity is typically zero, but positive and negative polarity can be either a set of discrete values, such as plus or minus one, or any real values, such as positive or negative.

Sentiment analysis or opinion mining has a wide range of applications. Among the popular ones are automatic ranking of products and services. We're familiar with rankings on Amazon, Netflix, car dealerships, and other storefronts. However, this is not always a centralized database or location where the opinions are stored. In a centralized

environment, these opinions can drive relative price and they can promote better products or they can help match buyers and sellers. However, many opinions are decentralized in tweets, blogs, video transcripts, yet this is additional and valuable information that companies are desperately seeking in order to improve their services and reputation. Note that external posts may need to be classified as "subjective" or "objective" before sentiment is measured. This is an additional layer of challenge in sentiment analysis.

In another use case, investment banks, hedge funds, and individual investors are looking for trading ideas arising from recent sentiment trends towards publicly traded companies, digital currencies, and other investment vehicles. There are many challenges in analysis and extraction of sentiment from text. Many models exist that can help address those challenges.

Back to Table of Contents

Watch: Sentiment Analysis Metrics

You can assign positive or negative value to a subjective text using sentiment analysis metrics such as binary polarity (a discrete metric) or valence (a continuous metric). In this video, Professor Melnikov explains sentiment analysis metrics, how to transition between continuous and discrete metrics, and he introduces the VADER lexicon model.

Video Transcript

A common task in sentiment analysis or opinion mining is to associate a subjective text with a positive or negative number indicating its polarity. Typically we use either binary polarity or semantic orientation or continuous polarity or sentiment intensity or valence. A binary polarity assigns a value plus or minus one to a positive or negative sentiment, but it can also use zero for neutrality or objectivity. For example, a noun "airplane" may be scored with a zero since the word lacks opinion. On the other hand, valence is a continuous number, but often is normalized to a finite range, say from negative one to one. We can transition between these continuous and discrete metrics if needed, or comparisons of different models. For example, valence scores can be thresholded so that all negatives collapse to minus one, all positives become plus one, and zero can be in either group or remain as zero. Or we can treat a small interval around zero as a neutral sentiment, collapsing it to a zero score.

To build valence scores from binary scores, we can rely on statistics of a sample, for example, the polarity of a sentence can be an average or a weighted average of all scores derived from the words in that sentence. Similarly, a document polarity can be an average of sentence polarities in that document. Valence scores are more flexible but require significantly more effort to compute and validate. Many lexicon-based models are human-curated, relying on linguists, psychologists, and other scientists to store sentiment of a word in a dictionary. Thankfully, these scores can be reused for building more advanced or specialized lexicons. That's how lexicon-based models evolve — they absorb and extend previous lexicons and standardize their sentiment scores. In some cases, the new words are added and scores can be collected through crowdsourcing.

For example, a popular VADER lexicon placed 10,000 new words, emoticons, and slang on Amazon Mechanical Turk platform where qualified raters all over the world ranked these words for a small fee of about $0.01 per word, plus a bonus for higher quality. Each word

received 10 integer ranks, each ranging from minus four to positive four, indicating sentiment intensity. A zero represented neutral, while other scores correlated with slightly, moderately, very, and extremely positive or negative opinion. These 10 ranks were averaged and only those with standard deviation of less than 2.5 where retained for the final lexicon, which ended up with 7,500 tokens, approximately.

In other situations, machine learning models can be used to rank words and sentences, but this turns out to be somewhat noisy, although easily adaptable to new domains. Also, if a word is stored in a taxonomy such as WordNet, then we can draw its sentiment score from its depth in the taxonomy tree — words deeper in the tree can indicate a greater specificity and stronger valence.

Back to Table of Contents

Watch: Introduction to Lexicon-Based Analysis

Professor Melnikov discusses a variety of lexicons that have a portion of their words with a sentiment analysis score, such as the Linguistic Inquiry and Word Count (LIWC), general inquirer (GI), and WordNet. He further expands to discuss SentiWordNet, which uses a machine learning model to score all words in WordNet that did not previously have a sentiment analysis score.

Video Transcript

In a lexicon-based model, a dictionary of sentiment-scored words, emoticons, and other lexical features is used to compute a sentiment for a document. The lexicons are often curated by experts and stored in the model itself. For example, Linguistic Inquiry and Word Count, or LIWC — pronounced as "Luke" — has 4,500 words in 76 categories, including 900 words ranked with a binary polarity as determined and continuously improved by leading experts in the field. General Inquirer, or GI, is a similar lexicon from Harvard University with 11,000 words in 183 categories, including about 2,000 words in each polarity. WordNet is another database with binary polarity for some of its words. These lexicons cannot distinguish between "good" and "terrific," but they are still popular because of their high quality and inference speed. They also suffer from a very limited vocabulary. For example, only a fraction of 150,000 words in WordNet are actually labeled with polarity, and 150,000 vocabulary itself is just a tiny representation of words in the English language.

SentiWordNet automatically extends WordNet synsets to include positive, negative, and neutral sentiment ranking, which vary from 0–1 and sum up to 1. Here, the scientists identified exemplary positive and negative seeds of synsets. Then they used a machine learning model similar to the nearest neighbors algorithm to score all remaining words by their proximity to either positive or negative or neutral clusters.

Back to Table of Contents

Quiz: Sentiment Analysis Terminology

In this module, Professor Melnikov has introduced terminology relating to sentiment analysis. Now it is time to test your understanding of it. Select your responses (keeping in mind that a question may have more than one correct answer). Submit your quiz when you have finished.

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

Please complete this activity in the course.

Back to Table of Contents

Watch: Conduct Sentiment Analysis with the VADER Model

Valence Aware Dictionary for sEntiment Reasoning (VADER) is a lexicon based model that ranks the sentiment of short sentences from -4 to 4, using an established lexicon and adding in slang and emoticons to the lexicon. In the following videos, Professor Melnikov delves into how to use VADER in code.

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

Introduction to the VADER Model

Here, Professor Melnikov demonstrates how to use VADER and how it handles words with numbers or punctuation. He further illustrates how using capitalization, word order, including “but” in a sentence, and changing punctuation affects the ranking.

Video Transcript

VADER, or Valence Aware Dictionary for Sentiment Reasoning, is a successful lexicon-based model that aims to rank sentiment for shorter messages, including tweets and social media posts. The authors added emoticons, acronyms, and slang to the established lexicon and then asked about 20 Amazon Mechanical Turk raters to score these tokens on the scale from negative four to four. Then the scores were averaged and filtered by quality. In addition, VADER uses several heuristics to adjust valences of words in a sentence based on capitalization, exclamation signs, negation, a presence of the word "but," and intensifying adverbs.

Let's try it in code. We're loading some of the packages that we will need. Also VADER lexicon is here. And the object that will allow us to interact with VADER package is SentimentIntensityAnalyzer, SIA. That object now gives us access to lexicon property or attribute, which brings a dictionary of pairs or words as keys and their scores as values. There are about 7,500 words or keys in this dictionary. We can wrap it into a DataFrame to nicely display it horizontally or transposed. DataFrame gives us the words ordered by their scores. You have "ilu," "ily," "aml." The original scores were from negative four to positive four, but as the ranks were averaged, the scores converged towards zero little bit, at least the extreme ones. We have some very negative words here on the right-hand side.

The distribution of these words and their scores is displayed as a histogram. We can see that it's a bimodal distribution just like we wanted. We would not want to contain a single mode distribution because in that case we would probably have all the words centered around zero, which would probably be most English language words lacking opinion or sentiment. We want just the highly polarized words to be in these lexicons that we can use to rank sentences and words.

In this next cell we're displaying words that have some sort of punctuation or numbers, and this turned out to be just emojis and some more complex forms that are often used in social media. We have 143, which is "I love you," "sweet<3," "8d" — this probably looks more like a happy face when "D" is capitalized — and "d8" has a very negative score.

Now, the application of this VADER to the sentence would return to us three different sentiment scores and one compound score, which is not directly built from those three — negative neutral, and positive — but is meant to represent those. The word "yes" is strongly positive, "no" is strongly negative. "Yes" with an unhappy face balances out to

be about zero. Then we have "good idea" and "GOOD idea" with a boosted adverb, or adjective in this case, in all capitals. We have "good idea!" with different exclamation. At some point, when you have too many exclamations next to an adjective the polarity just drops out. And I'm not sure what it is with the model, but be aware of the situation. If you put a space in between adjective or polarized word and exclamation or flip the words around so that adjective is on its own surrounded by spaces, then the model picks up the polarity and we're back in business.

"Not a good idea" is correctly picked up; the negation flips around the polarity of the word "good." "It isn't a good idea" is also picked up, which is not true about some other models that we will see later. "Good and risky idea" balances out the polarities of "good" and "risky." "Idea is good, but risky" intensifies the polarity of the word "risky" which follows the word "but."

Measure the Polarity of Movie Reviews Utilizing the Sentence Polarity Corpus

Follow along as Professor Melnikov retrieves the sentence polarity corpus, a corpus of movie reviews that each have a polarity of +1 or -1. He then uses VADER to rank the sentence polarity corpus and analyzes how well VADER is able to correctly identify the review polarity.

Video Transcript

Many years ago, Pang and Lee from Cornell University compiled this wonderful corpus of movie reviews with their polarities plus or minus one. It's called Sentence Polarity corpus. It has about half and half of positive and negative reviews that were drawn or retrieved from Rotten Tomatoes reviews. Each review is about one sentence long, lowercase, and we can evaluate what it has. It has two files, positive and negative, with the categories that correspond to those file names. We can retrieve these reviews and their labels, create labels for those reviews, and run them through the VADER. Each review would be run through the VADER package model. The corresponding scores are added to the DataFrame. So we have this DataFrame with all the reviews on the right-hand side, 10,000 of them, and the rankings that were derived. Number of characters is also added. The vY is the actual review or polarity that was identified by the authors, and the pY is the predicted. The predicted is just the thresholded polarity minus one picks up all the negative scores; plus one picks up all the positives. And zero — you can also threshold zero to be a separate score.

Now we can use metrics in Scikit-Learn to evaluate the quality of this model. We look at the confusion matrix. It suffers from lack of labels here, but we do know the diagonal values indicate the correctly identified labels and off-diagonals are incorrect labels that we want to focus on. A better representation would be some colored image with the true labels on one side and predicted on the other. So we know now that 30% of reviews that were negative were classified as positive. We have the opposite, about 9% being positive reviews that were classified as negative. And we probably would benefit more by focusing our attention on dealing with this larger percentage. Why were these misclassifications made? We can investigate with the actual reviews to see what text structure was picked up or which one was ignored.

Overall, the model performs reasonably well, 0.68 f1-score — remember it's between zero and one, higher value is better. For the 68% for the positive class and 51% for the negative class, the actual negative class underperforms, as we observe from the confusion matrix. If we want to improve the model or lexicon that it uses, we would look at these individual reviews. And it's probably convenient to filter out all correctly labeled and focus our attention on negative reviews that are classified as positive, ordered by the strength of the compound score. We also notice that the longer reviews are classified more poorly. At the top, we have most extremely poorly classified reviews. They are difficult, even for an expert, to gauge the ranking or sentiment from.

We do notice some of these reviews have parts of the movie title that are negative or positive words themselves, and they participate in the ranking, which is probably not something we want the model to do. So, "divine secrets of the ya-ya sisterhood is not horrible, just horribly mediocre" is a negative review for a movie that contains a positive word in the title. So we might, as a preprocessing, remove all the titles from the reviews so that something like "Great Gatsby" does not impact the sentiment. If we need to, we can add more words to the lexicon to improve the movie reviews. So somewhere here there are words or combinations of words that — even "horribly mediocre" could be added to a dictionary to improve the lexicon. There are some other ones that I've seen that would probably be ranked negatively in this particular context or in this particular domain.

Back to Table of Contents

Code: Practice Conducting Sentiment Analysis with the VADER Model

Previously, Professor Melnikov demonstrated how to use the lexicon based VADER model to determine the polarity of sentences. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice retrieving valence scores for various words in VADER's lexicon.

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: Conduct Sentiment Analysis With the TextBlob Model

The TextBlob model can be used for sentiment analysis, and is trained using a simple feature extractor which determines whether there are positive or negative words. Professor Melnikov describes TextBlob and then discusses the advantages and disadvantages of using TextBlob instead of VADER. Using Python code, he shows a comparison of sentiment analysis results from both models along with the aggregated statistics for a more complete comparison.

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

Video Transcript

TextBlob NLP library also contains a sentiment analyzer based on the predetermined lexicon and a Naive Bayes classifier trained on a movie corpus of reviews from NLTK library. The default model is trained on features from a simple feature extractor which indicates whether positive or negative words are in the document or not. Once the model learns the relation features to the document's binary polarity, the model is ready for inferencing. Interestingly, TextBlob classifier is twice as fast as VADER and performs approximately equally well in positive movie reviews, but significantly underperforms on negative reviews. However, TextBlob offers an easy interface to add your own feature extractor and retrain the Naive Bayes classifier on your own opinion corpus.

Let's try TextBlob in code. We're loading TextBlob and movie reviews in this cell. And the TextBlob object can take a sentence, and then we can use sentiment attribute and polarity and subjectivity attributes to extract those values. Polarity ranges from negative one to one, whereas subjectivity is a 0–1 interval. Here, "good idea" returns 0.7 and 0.6. We can try predicting the sentiment for the sentences we've seen before by individually passing them to the TextBlob object and retrieving the polarity. Here we have polarity and objectivity — why not? — and the words "yes" and "no" do not have any polarity whatsoever. "Yes" with an unhappy face has a negative polarity of negative 0.75. That means that emoticons are part of this lexicon. "Good idea" and capital "GOOD idea" do not differentiate, so there is no boosting for polarity due to capitalization, but there is boosting due to exclamation, although it's limited — I think it stops after two or three exclamation signs at 1 maximum value. But at least it doesn't disappear when you have too many

exclamations like happens with the VADER package. "Not a good idea" is correctly picked up as negative 0.35. "It isn't a good idea" still appears to be positively charged, whereas it is not. This could be problematic for some of the negative reviews. "Good and risky idea" and "idea is good, but risky" do not differentiate. So there is no boosting with the word "but" to the second phrase following it.

When we compare TextBlob and VADER, we can investigate a bit more, in greater detail the differences between positive and negative reviews as well as individual reviews that might be causing problems for either model. We can retrieve the movie reviews and we'll just take the 100 from each, not the full 1,000 from each reviews. There are two categories. Here are some examples of the reviews, and here's the application of TextBlob and retrieval of the polarity from each of the reviews. The polarities are then thresholded so we have actual polarity in the first row vY, or validation label. We have polarity, which is a continuous number from negative one to one, in the second row or column, if you think of it as the transposed matrix that we created. Then we have this predicted polarity; also negative one and one values are in it. We can see that the positive polarities are mostly predicted well, with an exception of two, in this displayed list of reviews, and negative polarities are predicted poorly; instead of all minus ones, we see a lot of positive ones in the bottom row.

If we do the same trick with the VADER package SentimentIntensityAnalyzer, we're retrieving the compound score, not the individual polarity scores. The compound score is also thresholded at zero, so we observe that the VADER performs slightly poorly or worse on positive reviews, but a lot better on the negative reviews. All the negative ones, or most of the negative ones are predicted to be negative one by VADER. The aggregated statistics can tell us a little bit more information. Although we only have 200 observations, we can still make some sort of inferencing. We have 0.70 f1-score for the positive reviews versus 0.68, and that's how it is expected. The VADER performs a little bit worse. But we can see that 0.34 for the TextBlob is a lot worse than 0.51 for the VADER. Of course we want this number to be closer to one, and it ranges from 0–1, higher number is better. And if we want to improve either one of these models or even both, we would focus our attention on the reviews that are misclassified and perhaps misclassified by both models. Like review number four is misclassified by both. We can investigate why this happens.

Back to Table of Contents

Code: Practice Conducting Sentiment Analysis With the TextBlob Model

Previously, Professor Melnikov demonstrated how to conduct a sentiment analysis using the TextBlob model and showed a comparison of results from the TextBlob and VADER models. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice computing and comparing metrics for the TextBlob and VADER models.

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 — Conducting Sentiment Analysis Using Various Techniques

In Part Three of the course project, you will write functions to expand VADER'S lexicon using TextBlob's sentiment analyzer on words that are missing from VADER's vocabulary. You will then measure the f1 score for both the initial and expanded VADER lexicon and compare the results. The tasks you will perform are based on videos and coding activities in the current module, but may also rely on your preparation in Python and basic math.

Resources

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

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

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

Completion of this project is a course requirement.

Instructions

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

Please complete this activity in the course.

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

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Conduct Sentiment Analysis Using Various Techniques

In this module, you were introduced to sentiment analysis, as well as various metrics for evaluating sentiment, including polarity and objectivity. Using the VADER and TextBlob libraries, you trained, tested, and evaluated two sentiment analysis models. Finally, you compared the performance of both models on a set of movie reviews.

Back to Table of Contents

Thank You and Farewell

Oleg Melnikov

Visiting Lecturer

Computing and Information Science
Cornell University

Congratulations on completing *Conducting Semantic and Sentiment Analysis*.

I hope that you now feel more equipped and empowered with the tools needed to identify taxonomies within a language, and apply databases of word attributes to solve various NLP tasks.

Although, this is the last course in this sequence, the field of NLP is far larger than the sequence can cover. You've built a grand foundation; and we encourage you to continue building upon these skills. Now, you can dive deeper into deep neural networks and learn to build or extend language models (such as **SBERT** that you used earlier) to solve a much wider spectrum of language problems. Now, you have a sufficient background to start learning about the new and exciting work with large language models (LLM), such as **GPT4** and ChatGPT.

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

Sincerely,

Oleg Melnikov