Eliezer Kanal: Hello, and welcome to today's webinar, SEI's webinar. Today we're talking about Secure Your Code using AI and NLP. My name is Elie Kanal. I'm a technical manager here at the Software Engineering Institute. Thank you very much for joining me. I'm joined today by Dr. Nathan VanHoudnos and Mr. Ben Cohen, both of whom are machine learning research scientists here at the SEI. Thank you very much for joining me, guys.

So there's an awful lot that we do at the SEI. Part of our overall mission is helping write secure code, helping generally to make sure that software is written in a secure way. One of the techniques that has been pretty popular in a whole variety of areas recently has been natural language processing, and something which we've been looking into is how we can apply that technique to writing more secure code. So we thought we'd start off a little bit with how natural language processing itself works as a practice. So, Ben?

Benjamin P. Cohen: Yeah, so if you're not familiar with natural language processing you've still probably come across it just in your daily life. So if you've dealt with customer service chat bots, autocomplete with your text messages, recommended replies through email, these kind of things, NLP is kind of what's driving that under the hood. So NLP at a high level is how machines extract information from language which is occurring naturally. So a typical pipeline for NLP will start with something like raw text. You could do speech recognition as well, but we'll focus on text. So you start with raw text, and then that text needs to be processed in a way that's easy for machines to work with. So you acquire a representation of that text, and then machines will basically study the data that you've given it and find sources of repetition, find patterns in it and stuff, and then once that process is complete, you can use NLP for a variety of tasks. So you can do things like predict what's going to come next, as in autocomplete; you can create naturally occurring sequences of language, which you see with things like chat bots; and so on.

So, NLP has been around for a hundred years or so, but it's really blown up in the last maybe decade and change. So the real reason this seems to be is because now we have large amounts of data that are available in electronic form, which didn't used to be the case, and that's afforded us a lot of insights and it's very easy for computers to work with that data. So given some of the successes that NLP has had recently, our question is, "Can we apply this to source code?" When you think about it, source code is really just a special example of a language. It's text, just like everything else is, and it is a form of language, and we're curious to see if we can figure this out and apply NLP to source code.

So, the reason that this is possible is because of something called the naturalness hypothesis. So natural language is very repetitive and full of patterns, so even though there are poets, like Mary Oliver, and playwrights like Shakespeare that use language in very sophisticated, nuanced ways, and language is capable of conveying incredibly complex concepts, most of the conversations we have on a daily basis are kind of straightforward and repetitive. The naturalness hypothesis says the same is true of source code. So if you can find the sources of repetition, that means you can predict what's next and make use of that information.

Eliezer Kanal: So once you have such a representation-- once we understand, sorry, that text and code in general is pretty repetitive, we have to now understand, "All right, so we have this text in front of us. How do we get a machine to actually understand it?" And you can think-- you were mentioning before we have these different types of text that are now publicly available. Project Gutenberg, for instance, has hundreds of thousands of books, and you can look through all that, but when you're trying to get a computer to understand what's in the text, it's not looking at the text itself. As it turns out, there are a variety of representations that the computer has to use to understand what's in the text. It's not just looking at the words as you and I do. So to that extent, we have this little chart here which shows all the different levels at which code can be understood by either a trained linguist or by a computer, and when you're trying to have the computer understand what it's looking at, it has to understand all these different parts of the hierarchy in order to be able to make sense of what it's looking at.

So you can start-- the very most basic, we have this thing called morphology. Morphology is the process of taking a single word and breaking down that word into its components. Some words, like "the" or "baseball"-- the word by itself stands alone and you don't have to do much to it. But other words you can think, such as "mustn't", or even "office building", there's context you have to add to it. You have to separate that into "must not" or whatnot. One thing which I demonstrate in this slide that you see here, there's a Hebrew word, tievgoshe [ph?]. That Hebrew word actually means three separate words in English-- "you will meet"-- and you can see the coloring that the other words also break out into separate ones, and depending on how you're working in context, what the language is and what the model is, you have to do this for all the words in a given sentence. Once I understand what the words themselves mean, then I have to understand how I'm applying-- which particular definition of that word should be used.

So as you can see, there is the word "bank". That word "bank" actually has multiple different possible meanings. I can be talking about a bank, the financial institution; I can be talking about a riverbank; I can be talking about banking an airplane; and I'm sure there's quite a few other ways that this word can be used. In order for a computer to understand a sentence, it has to be taught how that should be used in context.

Once I get past understanding now what the word is, then I can get into what we call syntactical understanding, and I have to understand the syntax of the sentence. Now I'm taking multiple words, now that I understand what these words are, and I'm trying to build them into an overall structure. One example which is commonly given comes from Groucho Marx when he was playing Captain Spaulding in "Animal Crackers". "One morning I shot an elephant in my pajamas. How he got in my pajamas I'll never know." That sentence comes across to audiences as funny, but to linguists, they actually break that down into the different sentence structures that it can mean and demonstrate quite clearly how this actually can be interpreted two ways by a computer. In that context, once the computer has this and sees these two ways, it now can understand, "Oh, here are the two possible ways that I can put these together."

The last one that's kind of at the level we're going to deal with is the semantic meaning. So now that I have a sentence and I understand exactly how all the words fit in next to each other and I understand what the words mean, I have to understand the intent, and that's where we get to semantics. So even if I know, "Remind me to buy groceries when I leave the house." Well, I can understand how the syntax and what "groceries" means relative to just goods, but I have to know now that actually means create a reminder, and when I do something, there should be an action. That is the semantic level, and that's a pretty complex task that we're finally able to reach-- I'm sure many of you listening are familiar with this-- using all the tools that we have at our disposal on our cell phones and whatnot.

The last two are kind of still in the future. We talk about pragmatics. Pragmatics is nonlocal meaning. So when I say, "Hey, please pass that down," that sentence may mean something or may not, but in context it's almost impossible to know what it means without looking at previous things I've said, the rest of the discussion and whatnot. So in order to really understand the pragmatics of a sentence, I have to put it in a much larger context. And the last one, which is the most complex, is discourse, where two people are having a conversation; there's multiple references that can go not only to previous things said in this conversation but previous conversations on previous dates, and trying to understand all of the context is a very difficult task. As it turns out, some of this stuff can also be applied to code as well.
Nathan M. VanHoudnos: Exactly. So here-- hopefully you can see it on your screens. Here is some JavaScript code that actually Elie wrote for this exact purpose to illustrate how some of these ideas can work as computers are asked to understand code. At the lowest level, we have symbols, or the morphology. For those of us who write JavaScript or use JavaScript, you'll know that curly braces are very important as a reserved word, such as "var" and "function". These are the lowest level of understanding that you could have about the code, is just the symbols.

Coming up a level, you actually need to understand the lexemes or the context. Here what we've highlighted are two usages of the word "time". In one case, it's the name of a variable. In the other, it's a string that talks about a different part of the document where it needs to get a particular attribute. And so being able to distinguish between time meaning variable and time being a string is an important distinction in code.

Moving up the hierarchy, we also have syntax problems. Well, we have syntax that we want to write good, syntactical code--

Eliezer Kanal: And frequently as programmers, we have syntax problems.

Benjamin P. Cohen: And I have syntax problems.

Nathan M. VanHoudnos: Ooh, I probably shouldn't cover my mic. Break the fifth wall there. Our fourth wall. Anyway, anyone who's written code has had problems with syntax. Your computer language will let you know about that pretty quickly. And above syntax, since this is actually a web application that we have demonstrated, we're actually using APIs interacting with other applications, so there's also an element of discourse as people write the code, even if it may be difficult to understand the code at this level.

And so this is a pretty specific example, but it's actually a really wide area of research, as people have applied natural language processing to code. There's a particularly good survey paper which we have up for you, "A Survey of Machine Learning for Big Code and Naturalness". It's sort of an NLP for big code paper. It's an excellent resource for the field. I recommend that all of you who are interested in this sort of thing go and read it. It covers things like code generation models, representational models, and other ways to mine patterns. We're going to be talking about a couple of these things through this webinar.

Eliezer Kanal: And I want to just take a minute to remind everyone who's listening here, we have chat available. Feel free to ask questions through the chat if you'd like, and we'll be able to answer that as we're going through. So why don't we see how it is that some of these techniques are being currently used to understand code.

Benjamin P. Cohen: Sure. Right. So like I said earlier, one of the main reasons that NLP has taken off is because we're able to find patterns and sources of repetition in data. That means you're able to predict what's next if you can find those patterns. So just an intuitive example here is, "For lunch today I had a peanut butter and jelly," blank. Chances are most of us intuitively answer that with "sandwich", and to kind of spell out the intuition for what's going on there is you have a vocabulary, a list of words that you can fill in the blank with, and maybe you're assigning kind of a probability or at least a feeling of how likely a given word is. So "sandwich" feels very likely.

So this data structure, like a sentence with a blank after it, has its own name in natural language processing. So these are called n-grams, and N refers to the number of words in the sentence. So a bigram is just two words. So this is like "jelly" and blank. What you'll notice about this is that the probability of "sandwich" doesn't appear super high here. So if I just saw this, I would assume "jelly", maybe "jellyfish" or something like that.

Nathan M. VanHoudnos: Or "jelly doughnut".

Benjamin P. Cohen: Okay. "Jelly doughnuts".

Eliezer Kanal: It depends on how hungry you are, how close it is lunchtime.

Benjamin P. Cohen: Yeah, now I'm hungry. Right, so you could have "jellyfish" or "jelly doughnuts". So that's with a 2-gram or a bigram. With a 5-gram, you've got more context, and so the probability of "doughnut" or "fish" goes down. The probability of "sandwich" goes up. And in the general case, you can have N words here.

Nathan M. VanHoudnos: Proudly speaking, I was going to say, so you can have sentences of virtually any length, right?

Benjamin P. Cohen: Right.

Nathan M. VanHoudnos: And no matter how long those sentences are-- it can be as small as, "I had a peanut butter and jelly sandwich." It can be the entire words of Charles Dickens. But at a certain point you want to say, "I can't really process all of that stuff." Right? "I want to only look at a certain amount of text, because I can't fit all of that in the memory of my computer, or I have other limitations. So let's just look at the last N words, for example." Might choose two, might choose five, might choose ten, and then you can see, "How can I affect my guess based on how many I go back?"

Benjamin P. Cohen: Yeah. So, absolutely. Kind of an aside here that's just sort of interesting to think about is that our language is really complex and this is kind of the reason that you can catch someone for something like plagiarism. Like the likelihood that two sentences that are really long and whatnot occur exactly in two places is really low. So in that case, it makes sense that one is actually copied from the other.

Eliezer Kanal: You're simply looking at the probability that these two sentences would be the same given all other things that are changing.

Benjamin P. Cohen: Right. Yeah, so these ideas all apply just as well to computer science. So here we've got a Python code for i in range 10, and one of the differences between natural language, like English, and source code like Python is just that the vocabulary, the sort of things that you can fill in the blanks with, is different here.

So first we'll kind of break this into chunks. This is called tokenization, breaking a sentence into individual vocabulary elements. So here we've got for i in range 10, and my programmer instinct is telling me to close the parentheses here, but if you just have a bigram, once again, it's not obvious what should come next, so maybe a plus or a minus seems likely. Four-grams, it starts to become more likely that we're going to close the parentheses, and so on.

One thing to note here is that this is language-specific. So in Python, you would close parentheses or you would follow with a colon, etcetera, but in other languages you could put other stuff in there. So an N-gram language model is likely to be language-specific even though it can be implemented for any language.

Eliezer Kanal: And we'll get to this a little bit more later as we talk about how you build a language model, but that's a very good point, and especially if you consider other English languages-- French, German, and particularly languages that have different types of punctuation that may come into play-- when you start talking about-- they call them conlangs, created languages such as Klingon, which has a whole lot of weird punctuation in it. People think I'm making a joke, but the language was written by a linguist who understands how languages are constructed and it would follow all these same rules, so you would be able to do the same sort of tokenization, understand what comes next. In this context here, Python allows a parenthesis to be a token, so you would want that to be something you'd think about and you'd be able to build the same type of language mode.

Benjamin P. Cohen: Right. Nice. Yeah. So we know N-grams work well as just a fundamental concept in NLP, so now the question is: Do these actually apply to source code? So the authors of the paper where the term "naturalness hypothesis" was coined, they actually tested this by grabbing some open-source projects. They grabbed ten open-source Java projects and they build N-gram models on random subsets of the project, and then what they do is, on kind of a chunk of that data that's held out to the side, they try and fill in the blanks and then they look at the real answer and ask, "How surprised is my N-model by what the real answer is?" So that's kind of how you can interpret the Y-axis loosely here, is how surprised the N-gram model is.

Eliezer Kanal: The lower it is, the less surprised it is.

Benjamin P. Cohen: Yeah. So lower is better. So with just one-- this is a unigram, so there's no context at all. This is just-- you're just guessing what word is going to be here. With two, this is like "jelly" blank. With five, "peanut butter and jelly" blank. So it makes sense that there's pretty much a monotonic increase in performance here, or decrease in surprise, if you will, as N increases.

The other thing that's interesting about this is they actually compare this to the English language overall using a standard data set, and what they find is that code appears to actually be much more repetitive and much more predictable than natural language.

Eliezer Kanal: Which makes sense, because the code itself tends to come from very limited vocabulary. It tends to come from a very well structured environment. We were talking before about syntax errors that we get. Whenever anyone's trying to program, they're always going to accidentally insert a colon somewhere where they weren't supposed to, or a comma where they weren't supposed to, and the language will complain to them, so they'll have to go back and fix it; whereas when I'm talking here, if I flub a word and accidentally say "sandwich" instead of saying, "Welcome to the SEI," I can keep on speaking and probably most of the people listening will understand generally what I'm trying to say. Computers are much more specific.

Nathan M. VanHoudnos: Right, and I'm sure all of us have made those kinds of flubs, which some of you have caught.

Eliezer Kanal: Exactly.

Benjamin P. Cohen: One thing I actually forgot to mention earlier, speaking of flubs, is that we're sort of in this age of big code. So with things like SourceForge and GitHub, there's just a total abundance of code that you can get publicly online. The authors of this paper point out you only need in the range of 62 thousand lines of code to get this type of performance here. So no matter how obscure the language is that you use, you can find probably ten times that easily on GitHub. So, not hard to get this level of performance.

And as a proof of concept, these authors sort of give a real-world example of how this can be used to benefit programmers. So they took the Eclipse IDE-- so interactive development environment, which is used for coders. It has a suggestion engine in it, so when you're typing-- you have "for i in range" something or other-- it'll give you suggestions as to what to type next. So to determine if N-grams can actually be useful here, they built an Eclipse plugin that acts on top of the code suggestion plugin that's already in place, and once again, what they find is that N-grams give a boost in performance here. So that just kind of hammers the point home, yes, code is in fact-- it has this property of being natural, and this is a real-world use case for N-grams.

So now we'll kind of pivot and move on from just talking about N-grams to other aspects of natural language processing.

Nathan M. VanHoudnos: So one of the things that N-grams are good at is they're good at predicting what comes next, but that doesn't necessarily mean that they have much understanding of what the meaning of the words are or the code is. So how do we actually solve this meaning issue, or at least begin to take steps towards it?

One way, sort of a traditional way to do this, is something called ontology, where you will actually write down all of the relationships between the words and all of these sorts of things that let us know that, "Okay, there are three different types of bank that we talked about." Right? All of this information. The problem with ontology is that they're very difficult to create and they're very hard to keep up to date once they have been created.

One of the basic problems here is that when we go from the words that we say or the words that we write down to something that the computer can actually process, the way that we encode that is often, as we've shown on the screen, where "hotel" will be a bunch of zeros and then it will have a one for the hotel column, and "motel" will be a bunch of zeroes and have a one for the motel column, and the computer, since it's only looking at those two differences of where the one is, it doesn't know that "hotel" and "motel" are actually very closely related to each other. They're just as dissimilar as "hotel" and "motel" and "TV". Or I guess "TV" is kind of similar. Maybe something like "pencil", right? Those are words that aren't similar.

Eliezer Kanal: You can think of it as essentially any entry in any index, right?

Nathan M. VanHoudnos: Right.

Eliezer Kanal: If I have an index of words, two words might be very similar, but they don't necessarily mean anything identical. It could be they're similar because they both fall under the same page, or they're both alphabetically organized, or, for whatever reason, the two acronyms happens to look similar. But there's no semantic meaning that's actually connecting the two of them.

Nathan M. VanHoudnos: Thank you for that. I appreciate that. So how might you actually do this? How might you actually learn this? Before we go to that, I'd like to give an example of how we just intuitively know about how words work. There's this wonderful quote, "You shall know a word by the company it keeps." Different words that are similar to each other are used in similar contexts. One of the techniques that we use for this is something called word2vec, and to be really specific, something like these two sentences-- "Usually the large-scale factory is portrayed as a product of capitalism," is one sentence, and "At the magnetron workshop in the old biscuit factory, Fisk sometimes wore a striped..." And it's the word around the word "factory" that will represent the word "factory". You will represent a word by its neighbors because that is a natural way to understand the meaning of a word, as opposed to try and come up with an index that is very specific, as we were talking about a minute ago.

So how might you do that? Let's think-- instead of words for a second, let's think about maps. So on the slide what we have is we have Atlanta, Chicago, Denver, all of these different cities in the United States, and then we have their pairwise distances between them. So we know the neighbors, very generally, about each of these cities. Then there's a technique called multidimensional scaling that you can use that will then figure out where those cities are in relation to each other, and if you just look at that, you can see that we have a rough approximation of the shape of the United States just by looking at the pairwise distances of cities. So if you can know a city's location but the neighbors that it keeps, you can also know a word's meaning by the neighbors that it keeps.

The way that word2vec does this is it in some sense makes a map of all of the different words that you have trained it on. So in this example, you can see that we have "man" and "woman", "uncle" and "aunt", "king" and "queen", and the path to go from "man" to "woman" is the same as the path from "uncle" to "aunt", and same as the path from "king" to "queen". It's the same, in some sense, compass bearing that you would take and then distance that you would travel. And when you have an encoding of the words that obeys those kinds of properties, you can do really interesting things. For example, you could say, "Okay, how do I pluralize a word?" Well, let's go in the plural direction, from "king" to "kings" or from "queen" to "queens", or we can even do more complicated things like analogies such as "king minus man plus woman" would then equal "queen", and it's these sorts of embeddings that have really moved the field of natural language processing forward.

If you want to know more about how this works, there are a bunch of different sites where you can play around with these things. We have them listed for you. There's also a variety of other papers that have continued to move this direction beyond word2vec to other directions such as taking into account not just the neighbors of the words in a sentence, but also, for example, its context in a paragraph and models such as that, and we can also even do this for code, which we'll talk about in great detail a little bit later.

Eliezer Kanal: Before we jump off that topic, it's worth mentioning, for those of you at home who are software engineers and know how to play with code, these tools, which you saw the links in the previous slide-- they're actually publicly available. You can download toolboxes that enable this functionality, and they're very, very powerful. They can really-- there are many tutorials online that will teach you how to use these tools and techniques to directly build all sorts of chat bots, all sorts of textual analysis work, and similar stuff, and it's a very powerful technique that lets really anyone who understands how to use code and how to interact with these APIs, they can build some very powerful tools. So it's definitely worth checking out.

So we've talked for a little bit now-- we've kind of referenced indirectly this concept of a language model, that I have to take code and put it into this representation that the computer will understand, right? So how do I build a language model? We went through what the computer has to understand to go into that, but how do I actually build it? Well, we can start looking at this to see what goes into a language model. So one possible sentence that I might have for building an English language model is something as the following on the screen. "Roethlisberger is a better quarterback than Brady." Arguably a completely true and not at all controversial sentence.

Nathan M. VanHoudnos: Indeed.

Eliezer Kanal: Of course. That should be said. Software Engineering Institute is in Pittsburgh and we are proud members of this fine city. But there are other sentences that you might not come across that wouldn't be present in any of the data sets that you're looking at.

One fairly famous example, which probably breaks this because it's a famous example, is one called, "Colorless Green Ideas Sleep Furiously." This particular sentence s interesting because it happens to be syntactically correct, but putting that aside, you'd never see this anywhere. You would really never see this in any context whatsoever. So because of that, the likelihood-- talking before about how surprised am I to see each of these words next to each other, ideas don't have color. Something that is colorless doesn't have a color, and then ideas don't sleep, and one doesn't tend to use the word "furiously" to describe sleep. It's just not-- these things don't go together. So when you're building a language model, you're simply looking at what is next to each other.

The same exact concept exists in code. We used this similar example before, but what you see on the screen now is a very common example of really run-of-the-mill code you'd see in any language in any codebase. Someone probably has something vaguely like this somewhere in the codebase. That said, what you see on the right-- actually, I don't even know, stage right, stage left. What you see on the other side there, the symbols themselves may show up many times. It's possible to have a var declaration. It's possible to have that little percent sign thing. There's good reason why you might want to use all those, but you definitely wouldn't see them in context. Building a language model simply says, "Look at what comes next and then just remember it," and it's remembering all of these statistical closeness that we're trying to encode in our language model to say how likely is it to see this next thing. N-grams has one approach; embeddings has another approach. There are many techniques for doing this, but the end goal is, "How likely is it to see whatever I have in front of me compared to the stuff I've seen elsewhere in the text, preceding or following?"

It's worth noting that when you're building a language model, it is completely and entirely dependent on the training data. So if you're going to be training a language model on, for example, Shakespeare, you can intuitively understand you're not going to be able to replicate Beyoncé lyrics, simply because those two data sets are completely different. If you're training a language model on Java code, it's very unlikely you're going to be able to replicate Python or Fortran or something different, because these are not the types-- the ways that languages are constructed are very, very distinct. The practical implication of this is that the training data that you're using for whatever your application is, is absolutely critical, and if there's a certain things you're looking for, or a certain very important aspect you want to find in your data that isn't present in the training set, you're not going to be able to find it using whatever model is built from that training set. You'll need to get yourself a better data set for that purpose.

So once I have this model and I'm able to build this thing and I have this model that was constructed from all the symbols I have, the code from all the data sets that I've been able to put together, what do I then do with it? Well, going back to the very beginning, I understand what I should be seeing in my data set, because after all, my training data set contained all this stuff, and I'm looking at my new data set to understand does it look similar to the training one. I can look to see, are certain things frequent? Is there a certain symbol, function command, or if I'm using spoken language, word, phrase, that I find very frequently in my training that I don't find in my test data set? I might ask, "Here's an example of some code that I wrote. How similar is it? Does this happen frequently?" I might want to say, "Given some non-code input," if I'm working with code again, maybe I have comments somewhere. Do I see other similar comments next to other things in my codebase? I've now built this huge statistical model of the data that I have in front of me and I just want to query and ask as many useful questions as I can from that.

So, once I've done something like that, I can now actually put this to use to try to actually build something and interrogate my code.

Benjamin P. Cohen: Right. So code2vec is a paper that's done pretty much exactly that. There are kind of two pieces to the code2vec scholarly work. There's the goal, which like the title suggests, is to get a vector representation of source code that's useful. Nathan gave the analogy earlier, in natural language processing you can have things like the vector for king minus the vector for man plus the vector for woman equals queen, or at least is most similar to queen. The goal with code2vec is to get vector representations of code that have similar properties. So two functions or two methods that basically serve the same purpose but are just implemented differently should intuitively point in kind of the same direction here.

So that's the goal of getting a vector representation of your code. The way that you do that is you have a task that you train a model to perform. In this case, the task slash motivating goal is to be able to predict a method's name-- a Java method specifically-- so a method's name given the body. The name of the method serves as just kind of a rough description of what the function or method should do, and it's worth noting that this task, predicting the name of the method, is one of many possible choices, and the task that you choose kind of drives what the embeddings are that the model learns.
Eliezer Kanal: I'm going to interrupt here for a second, because we actually had a question, which we had actually taken out of the slides earlier, and I answered next slide on here. But to

put the question out, Nathan, when you used the MDS example of the cities, it's noted here in the question, the distance between the cities is very easy to understand. We can understand that geometrically. It's two-dimensional space-- just how far is it between them. When you're doing this with words though, the concept of distance is much more complicated.

Nathan M. VanHoudnos: Yes.

Eliezer Kanal: Is there any way that you can explain how we can compute the concept of distance for words. I know you're about to get to this for code, but I wanted to get to this for words first.

Nathan M. VanHoudnos: Sure. So help me out on this, but off the cuff, what I would say is that when we had that example where we had "hotel" and a bunch of zeros and a one, and then "motel" and a bunch of zeros and a one, those are just two vectors. What we'd like is we'd like to train those vectors, instead of being always just a change of one apart, maybe they're a small change apart. So the simplest way to do it is to start with a coding that's similar to that, where the words are essentially arbitrarily far from each other using something like cosine distance, and then as you look at more and more examples of words, you then move "hotel" and "motel" closer together and "pen" and "pencil" closer together, until you begin to satisfy these relationships.

Eliezer Kanal: Yeah. To kind of extend upon that a little bit, so when we're dealing with the concept of distance with cities, it's very easy to visualize. It's in real space, right? So I just go from here to there, and I can picture this two-dimensional walk. When we're working in two dimensions, where we're just traveling across the globe, we understand intuitively how to move and what a distance means. It gets a little more complicated though when you have something which is a multidimensional data set, and that word seems really complicated, but actually let's make it really simple. If I'm trying to describe human beings, let me give you an example of some things I can use to describe them. We have gender, which more and more commonly is actually a nonbinary feature. We can have height. We can have weight. We can have eye color. We can have whether or not they have hair. We can have their age. We can have all sorts of funky descriptions about how that individual can be described. Well, I just listed a whole bunch of things, and that's not just two. So we can kind of create a concept where this person with all these features and this person with all these features are similar or dissimilar.

You can imagine someone who is identical in every single feature except they have a different eye color is more similar to someone than a person who has a completely differing set of features. One good question which you may ask from that particular example is, "Is that a good distance function?" It could be that even though everything is very different, those two people happen to have things very in common. For example, they go to the same university. The even have different background languages, but they still go to the same university, for example, so they may be very similar, which is an example of how you have to choose your distance metric very carefully.

The example that we chose earlier, talking about the code2vec, it happens to be that the distance function there that they use relies on an arbitrarily high feature vector. So there's a whole lot of dimensions. How do they find that? The math gets really complicated when you try to explain that. I really do refer-- the slides that we have here are available and there are a few links on the slide that Nathan was showing earlier that you can go and-- there's some walkthroughs regarding how this works. I don't want to try to get into too much detail here, but suffice to say, they take each of these words, they put them in his arbitrarily high dimensional space, and then, as Nathan was saying, they'll start to nudge them together. So "hotel" and "motel" will over time get closer together, whereas "hotel" and "TV" might get a little more separated. And then concepts such as freedom will end up in a completely different area of this overall space altogether.

Nathan M. VanHoudnos: Right. And just to sort of play with that a little bit, it's been noted that when you, for example, learn a language model in English text, you'll be even able to get high-level concepts. There'll be, for example, directions that correspond to gender or to race, and when you use these things in larger context, you need to make sure that your machine learning system doesn't, for example, learn different biases that are present in your training data, and all of that even begins to show up in these embedding models. So these things really do begin to encode a lot of information about the text that they're presented with, even in ways that you might not expect.

Eliezer Kanal: So I kind of made that shift to talk about how we can get these distance metrics in language, and I interrupted as you were about to describe how we can do these distance metrics in code.

Benjamin P. Cohen: Those are all good points, I think. So bringing this back to code2vec, as both of these guys pointed out, this is an empirical process. So sort of the distance between two pieces of code in a vector space that's acquired by statistical model-- that's dependent on the training data here. So the way code2vec works is you parse the code in some capacity; the code is then passed through a neural network, and the network is trained to make predictions. It's trained to predict the name of a method given the method's body. So to the extent that the model is successful and starts making better and better predictions, the embeddings for the methods become-- well, they're just moved around in such a way that code that does similar things should point in similar directions. So it's an empirical process and it's heavily dependent upon the training data.

So I'll kind of back up a little bit here and explain what an abstract syntax tree is, because this is what code2vec takes advantage of. Like I said earlier, you need a representation of code or of language that's easy for a machine to work with. So the idea here is to use one of the first layers of code compilation. So the abstract syntax tree is just another representation of source code. There's a one-to-one mapping between a code snippet and its AST, abstract syntax tree.

And the way code2vec works is it looks at paths between two variables. One way to intuit ASTs is that they're kind of similar to sentence diagramming in English, and if you stare at this for a while it sort of becomes intuitive, but you can look at-- for example, there's variable 1, variable 2. Let me back up one slide.

So this is the abstract syntax tree for a function that just takes two numbers, adds them together, and squares them. Code2vec will look at paths between two terminal nodes in the tree. So you've got variable 1, variable 2, and then you've got the path that goes between them. So code2vec will extract a bunch of paths from a piece of code, and that's going to look something like this. You've got a bag of paths or a bag of words, and this is a common representation that's used in NLP.

So each of these words actually gets a vector. Those vectors are used as input to a neural network. Neural network trains to make predictions about the name of a method. So there are these parallels to what Nathan said earlier about sort of a plurality access, like going from "king" to "kings" or "queen" to "queens". There are similar ideas here with code. So login ends up being most similar-- the vector for login ends up being very similar to the vector for logout, and so on. So there are a few more of these. There's download, receive, send, and upload. Yeah.

A; Just to pick up, there's actually a few more questions coming up on the chat here, and again, if you guys have more stuff, feel free to bring it up. One of them here is asking, "How does the code estimate false-positives and negatives?" So one thing which I think we didn't discuss much is-- we mentioned this is trying to predict method names. Could you talk a little bit more about how the code makes sure it's getting the right names?

Benjamin P. Cohen: That's an interesting question. I mean, the notion of false-positive and false-negative is slightly trickier here because-- so code2vec builds up this vocabulary of a bunch of method names that it's seen. So this is not a binary classifier where there's a simple notion of false-positive and true-positive. There's being correct or incorrect for each of these names in the vocabulary. So there's a false-positive rate for the function sum and for find and so on. Sorry, what was the question again?

Eliezer Kanal: To build off of that, I guess what I would say is this code is-- we have a method name that corresponds to a very specific abstract syntax tree, right? And once I have that abstract syntax tree that describes this method name exactly, I can then look at abstract syntax trees or ASTs for other functions and see how much do they match, and it won't be a binary match, but it will be a generally getting closer or really dissimilar. So when you're trying to calculate-- what was the word that I was looking at here? False-positives and false-negatives and all that stuff, how that works-- I don't know, could you just talk a little bit-- how would I do this when I have this sort of fuzzy logic rather than a very binary answer?

Nathan M. VanHoudnos: So, if I may, there are sort of two things going on here. One is that if you are trying to predict the name of a method, in the original code2vec paper what they essentially say is-- they do some normalization of the method names to make sure that you can at least get close to it and you don't, for example, worry about capitalization or things like that. And then they basically judge themselves on either they got it a hundred percent correct, or they missed it. So as we sort of have looked at this in our own evaluation of this work, we've seen that it actually does really well at exact matches. Additionally work by those same authors is the code2seq paper, where they will actually not just try to predict a method name itself, but actually the parts of the method name, like Upload This File, or something like that. They would break that to "upload" to "this" and to "file" and predict that sequence in the name. This is a common thing in Java where they'll use camelCase to separate concepts in the method names. So I hope that helps to answer the question at least a little bit.

Eliezer Kanal: Sure. There's another question asked earlier which was fantastic, which we should have gotten to earlier. "Can one gain an understanding of the code slash data structure or metadata from the compiler? And isn't the compiler doing some of this work as a side-effect?" So one thing which we didn't really talk much about here--

Benjamin P. Cohen: That's a great question.

Eliezer Kanal: Yeah, we're using all this fancy natural language processing techniques to understand code security. We have compilers; we have static analyzers; we have dynamic analysis. We have all these techniques that coders have been using for years. So I guess how do these two things fit together?

Nathan M. VanHoudnos: Sure. So to just sort of walk through the Allamanis paper, the Survey for Big Code and Naturalness, there's essentially this hierarchy of how a compiler works, right? So a compiler will look at the source code; it'll tokenize it; it'll parse it; if you're in something like C, it'll expand out the macros, do all of that kind of thing. Then from that, it will build the abstract syntax tree. Then it'll figure out control flow. It'll build an intermediate representation, and then it will optimize that intermediate representation and then produce some kind of object code, whether it's a portable executable or whatnot, depending upon what platform that you're on and what language you're in. Much of the early work was at the beginning of that compiler chain, where we were worried about just the tokens themselves, in the same way that much of the early NLP work was worried about just the N-grams and just the tokens themselves.

As we've sort of matured as a field, we've begun to come closer and closer up that compiler hierarchy, so now we're at the abstract syntax tree level, in the same way that we do sentence diagramming and those kinds of things in natural language processing.

Benjamin P. Cohen: I'd say maybe it also depends on the task too. So you just laid out there all these layers of the compiler toolchain. It depends on what you want to use it for. So if you're trying to just predict a method's name, maybe the abstract syntax tree is a good level to work with, but there are other levels that you can work with that may be better or worse depending on the task.

Nathan M. VanHoudnos: Right, not to mention whether or not you're starting with source code or you're actually starting with an executable binary file that you then need to infer the source code from. That's an entirely different direction.

Benjamin P. Cohen: Right. So you could work backwards too.

Nathan M. VanHoudnos: Yes.

Eliezer Kanal: Cool. So now that we've kind of gone through all the stuff that this code can do-- we've essentially given a 45-minute whirlwind intro into both the field of natural language processing and how you can use natural language processing to understand code. Now we can get into a little bit of-- the actual title of the webinar was "Secure Your Code Using AI and NLP". Let's get into that part.

Nathan M. VanHoudnos: Right. So if you have a machine learning system that can actually understand a little bit about your code, one of the first things that you might want to do is to just use that to clean up your code a little bit. And when I mean clean code, I mean things like coding conventions, right? Code that's written by one organization might be different than code that's written by a different organization, and if you're a new programmer, you want to be able to get up to speed on the codebase; you want to make sure that as you write new code, it looks like the code that your peers are writing so that everyone is on the same page. It makes code easier to navigate and to maintain. And so there's a whole line of work on learning these kinds of coding conventions so that we can actually use them to improve our ability to write good and maintainable and trustworthy software.

The way that these folks did it-- it was, again, Allamanis's group-- is they would take some code that they wanted to review and then they would change that code in a variety of ways using essentially some standard techniques, and that would give them a list of candidates for the possible completions of that code or the possible cleaner versions of that code. And then from those candidates, they would actually compare them to the language model that they learned from all of the other source code that's written by that company or written in that project, so on and so forth, and they would score the different candidates and then they would find out what the suggestions would be. And so, for example, where we have this here, where we have this little test class and we're trying to figure out a suggestion for the parameter "each". The Eclipse plugin that they wrote will actually give them some decent suggestions, such as test class, or so on and so forth. So you actually can use this understanding of code for relatively easy lifts, and as the field has matured, we've been able to do it for more and more complicated tasks.

Eliezer Kanal: And what the field of software engineering has come to identify is that clean code is really a prerequisite to well-designed code that doesn't have bugs, for a whole variety of reasons. If you have code that is written in a way that others can easily read it, if you have code that is written in a way that others on the team can easily contribute to it, you are much less likely to introduce new bugs during the development process, and other people who are checking it are more likely to be able to read it and actually find where you messed something up. But that said, we may have to worry at some points about how-- what if actual bugs do get introduced.

So it turns out there's a whole lot of things that we can work on with the techniques that we're describing here. One of the easiest things which we can try to do is find the bugs themselves. By applying these techniques, you can actually say, "Is there a bug in my code?" Looking at the language models and all the other, and we'll discuss some of those as we get forward.

Another one which we can try is have the computer automatically write secure code for me. We kind of understand at this point what secure code looks like. We've seen a lot of it. Maybe we can just have the computer create it for us. It's good at automation. Another thing we can have the computer do is say, "Well, here's code that was written. Let me describe to you what I think it should be doing." That can be useful just generally to describe it, as the documentation explains what it is, or, if I read the documentation and it's incorrect, maybe the code is written in a way that it's not doing what it's supposed to be doing.

And of course, lastly, any good piece of code should most definitely be able to brew me a cup of coffee. If it can't do that, just return it back to the shop or something. But seriously, moving on with that, we're going to touch on a couple of these right here-- finding the bugs themselves, and creating good documentation for that.

So, when we're trying to bugs-- to create code-- we don't want to create bugs. We're trying to create good code.

Nathan M. VanHoudnos: Indeed.

Eliezer Kanal: Yeah, seriously. Whatever. I believe one of us here has-- you had a sign in your office, "Creator of Technical Debt," which is--

Nathan M. VanHoudnos: Don't tell that to everybody out here.

Eliezer Kanal: Essentially a way of saying, "This is what we do. We just build code that probably has problems here."

When you're looking at all these language models and you're looking at all the code that you've ingested, there's some assumptions you can make. One of the assumptions you can make is that most of the code that's present in this language model, in the training data you've built your language model from, that code is probably correct. Why? Because it runs. Right? If it was mostly incorrect, it probably wouldn't even compile, let alone actually run. Given that, we can infer from there, if most of the code is right, the bad code is probably pretty rare. Using that thought process, if you see rare code that looks moderately similar to correct code, it's probable there's a bug.

We can bring this back to the previous example we had before. If I have a sentence that says, "peanut butter and jelly" something, most likely it's "sandwich". If someone wrote "peanut butter and jelly bathtub," that's probably wrong. If someone wrote "peanut butter and jelly doughnut", that's unexpected, and it's probably worth a second look, because who knows whether that's right or wrong, but really it doesn't look so correct because I'm used to seeing the word "sandwich". I should probably check it out. This concept has actually been used by researchers to build-- the built a tool that does exactly this. It'll look through the historical code you've already written and try to find areas where it looks almost exactly like other code. Does the tool know whether it's correct or incorrect? No. But the tool is able to compare every piece of code to all the other code you have and say, "This one looks really similar but it's a little different."

One of the things which they pull out here is this particular piece of-- or two pieces of code. They were able to read through all the code in this piece of software called Apache Pig-- it's actually a very useful tool for data ingest-- and they were able to identify, if you look at the A section, the part which is in red, there's a word there, "value". That "value", as it's written in this piece of code-- and if you don't read code, don't worry about it-- that word "value" is written in such a way that there's a vulnerability, and a malicious actor could cause this code to do something that the person who wrote it probably didn't intend. Because we were able to look at other code which looks very similar to it, the tool that these authors put together was able to autocorrect it down to what you see in B, where it put this two-string function around that declaration, thereby fixing the same problem.

There are other techniques that do something fairly similar. The one that we just saw here a minute ago uses N-grams. It's part of the way that they build their model. There's another technique called a deep belief network. Deep belief network, as far as we're concerned, we can call it some form of magic. It looks at all sorts of context around the words and does something very cool. Read up about it on Wikipedia. But the short of it is, instead of using an N-gram, I can use this deep belief network, and that will allow me to build out another type of predictor that can also identify, "Is this code similar to other code? And if so, what should it look like, or which parts do I think are weird?"

One of the ways they motivate this example of using this DBN-- look at the two sections of code on the left and on the right. The tokens are identical. The exact same words show up. The only thing different is the order. Their motivation is that N-grams would have a hard time with this because the tokens are almost exactly the same. So when they're putting them in order, almost all the tokens would appear even in the exact same order as shown, but the function is highly different, and it would be hard for the N-gram case to identify that there's a distinction between these two pieces of code, whereas the other one can do it very well. The built this whole technique where they use the same abstract syntax trees that we talked about before, the same ASTs, and they're able to convert this piece of code, or these two pieces of code, into these ASTs, use that to identify what's similar to each other, and again, they can predict what is appropriate, what is inappropriate, simply based on how much does it look like known code.

Nathan M. VanHoudnos: Right. So we've talked a little bit about getting our AI system to do code completion or to write our code, to write secure code. What about that thing where we were almost joking about writing documentation? Programmers hate to write documentation. It's one of the reasons that I am outed as the creator of technical debt.

Eliezer Kanal: Sorry.

Nathan M. VanHoudnos: It's okay. You are in fact my boss.

Eliezer Kanal: Bad boss.

Nathan M. VanHoudnos: No, it's fine. And so how might we in fact automate the writing of documentation for us so that we can be lazier coders and actually do more of what we would call work? So in terms of what you might want to do is you might want to make sure that that system is accurate; you might want to make sure that it's fast; you might want to make sure that it can be tied into your commit hook so that it's automated-- all of those kinds of things.

So how do they do this? At a high level, what they're using is they're using this technique called statistical machine translation. And how does that work? That means that you learn a language model for one language and an intermediate representation that that language model can use, and a language model for another language, and then you can come through one language model to another intermediate representation through the language model and back out, so that I could, for example, speak to you in English and then it could get translated into a different language such as Spanish or Chinese or what have you.

And so the way that this particular system works, to move through this quickly, is that if you have the Python statement at the top of "if x modulo 5 double equals zero colon" you'll have a language model encode that, and then you'll have the intermediate representation translate that into this other language model, and then you'll walk that all the way back down so that you can get the example of "if X is divisible by 5".

As you might imagine, this is sort of impressive, or at least I'm impressed by this, but it's also pretty limited. When we talked in the very beginning of the talk, we talked about the different sort of layers of natural language processing. The language models that we currently work at are only at the bottom rung of semantics, so that means that as we go from one piece of code up through a language model and back down through another language model, we have very limited understanding that's possible, and so the descriptions that are generated are very pedantic and it misses the big picture. There's a lot more of this work that's described in the review paper, but I'll pass it to you, Ben, to sort of wrap us up.

Benjamin P. Cohen: Sure. Yeah, we're about at time, so thanks to everyone who tuned in for joining us. Hope you learned something useful from this. So just to kind of overview what we've done, we've kind of covered two ideas in natural language processing. We've talked about N-grams, we've talked about embeddings, and then we've spent a few minutes going over applications of those techniques in secure coding. So like I said, we're at about time here, but if you'd like to continue the conversation offline, please feel free to reach out and contact us. So, yeah.

Eliezer Kanal: Thank you very much.

Nathan M. VanHoudnos: Thank you.

VIDEO/Podcasts/vlogs This video and all related information and materials ("materials") are owned by Carnegie Mellon University. These materials are provided on an "as-is" "as available" basis without any warranties and solely for your personal viewing and use. You agree that Carnegie Mellon is not liable with respect to any materials received by you as a result of viewing the video, or using referenced web sites, and/or for any consequence or the use by you of such materials. By viewing, downloading and/or using this video and related materials, you agree that you have read and agree to our terms of use (http://www.sei.cmu.edu/legal/index.cfm).
DM19-0416