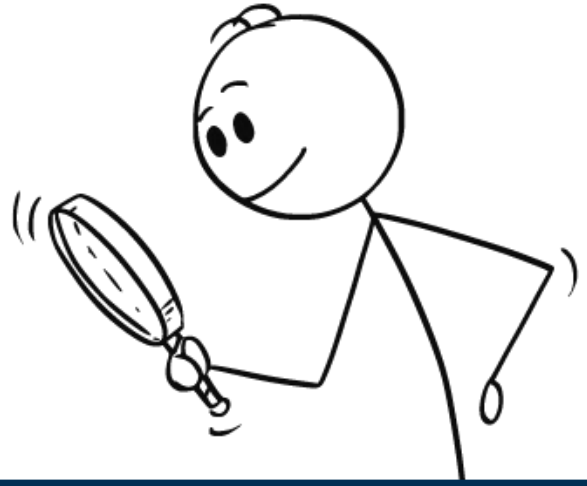


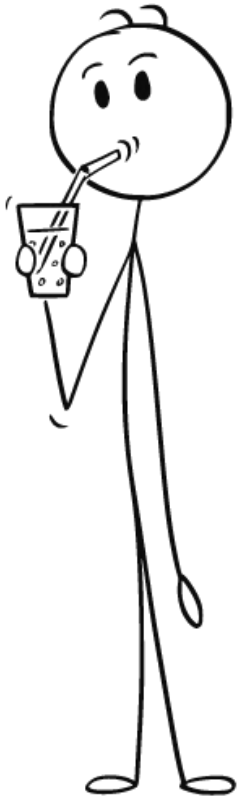
CSE 4/535
Information Retrieval

Sayantan Pal
PhD Student, Department of CSE
338Z Davis Hall

University at Buffalo
Department of CSE

Before we start

1. Project 2 grades will be released by this week, keep your VMs running
2. Project 3 released
   a. will be discussed today

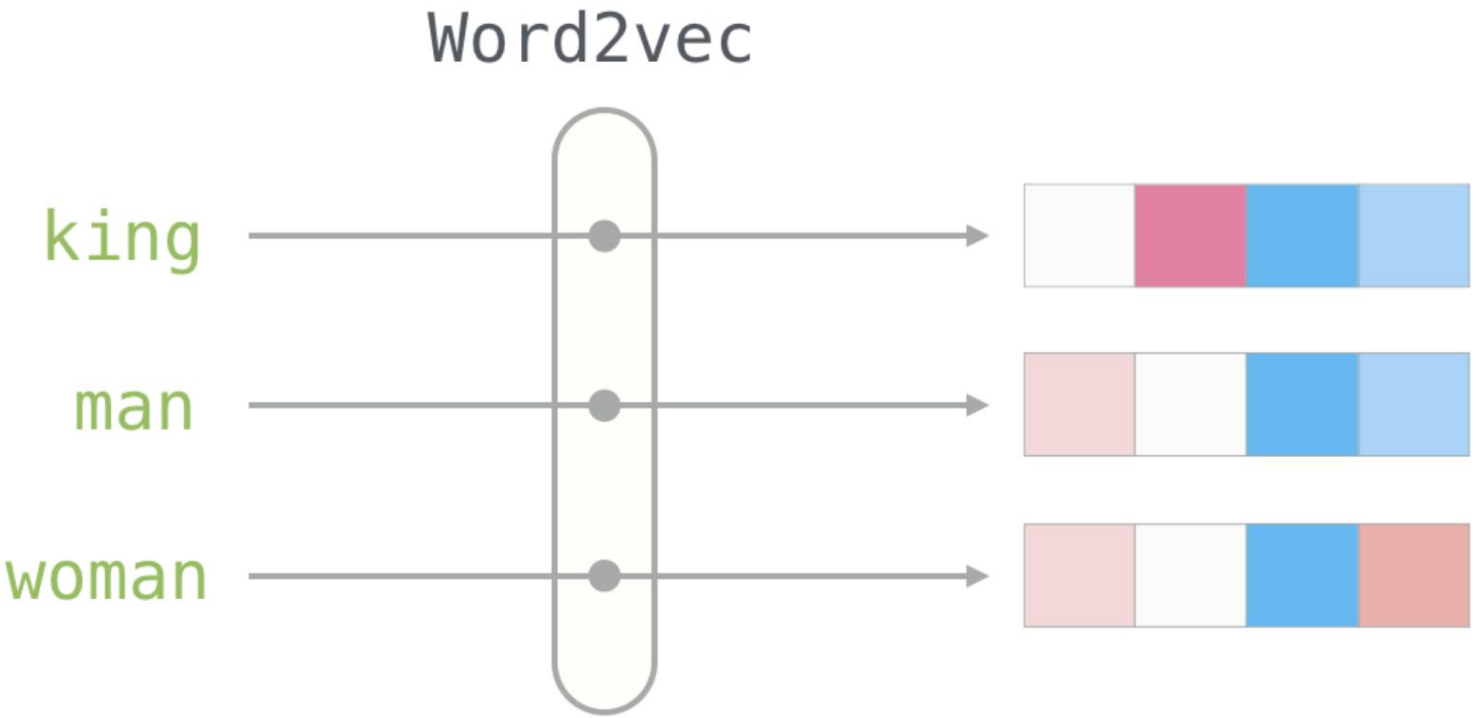


What are we learning today?

1. Embeddings - Credits to Jay Alammar
    a. What is Word2Vec?
    b. How Word2Vec works?
    c. How does similar words end up having similar embeddings?

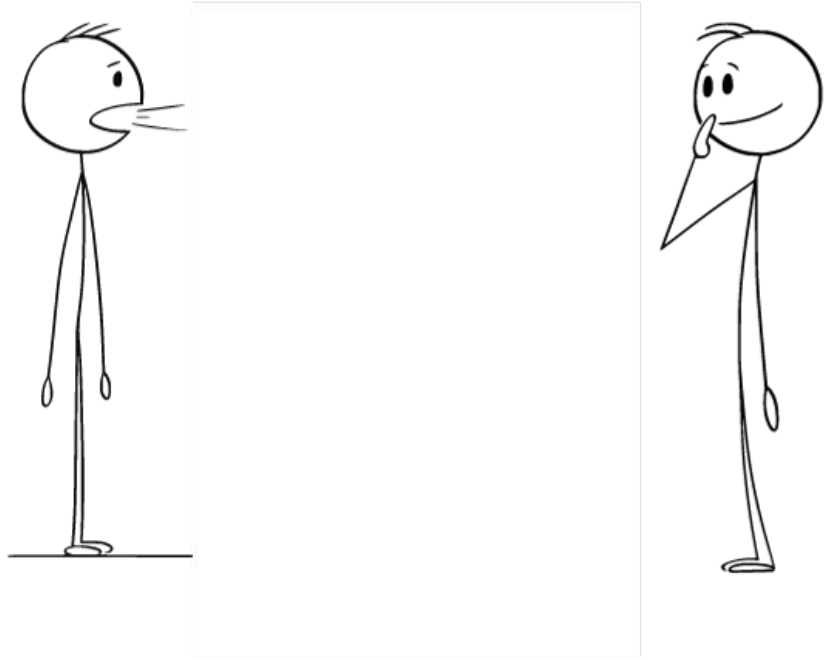
Let's start with project discussion…

Word2Vec



Personality Embeddings: What are you like?

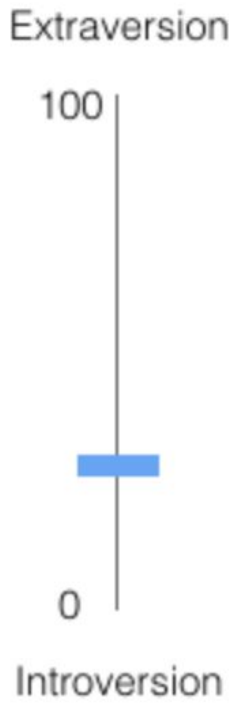
- On a scale of 0 to 100, how introverted/extroverted are you
  - (where 0 is the most introverted, and 100 is the most extroverted)?
- Have you ever taken a personality test like MBTI – or even better, the Big Five Personality Traits test? If you haven't, these are tests that ask you a list of questions, then score you on a number of axes, introversion/extroversion being one of them.

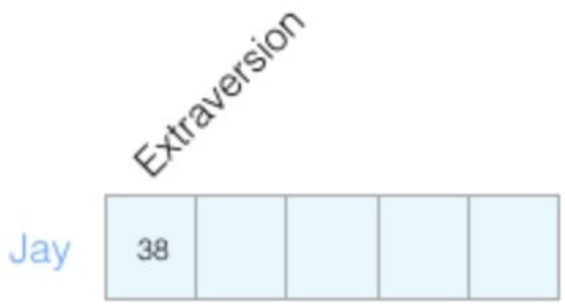


Openness to experience	79 out of 100
Agreeableness	75 out of 100
Conscientiousness	42 out of 100
Negative emotionality	50 out of 100
Extraversion	58 out of 100

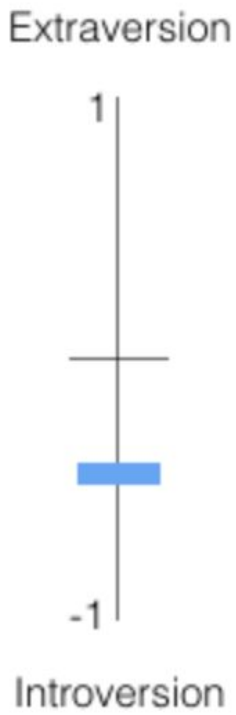
Example of the result of a Big Five Personality Trait test. It can really tell you a lot about yourself and is shown to have predictive ability in academic, personal, and professional success. This is one place to find your results.

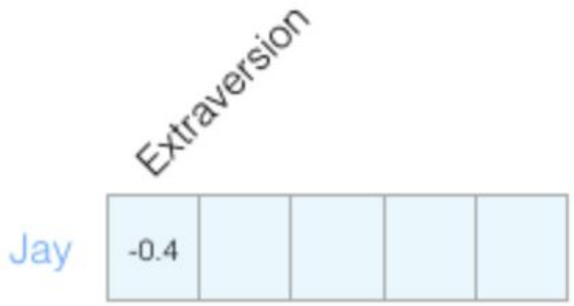
Imagine I've scored 38/100 as my introversion/extraversion score. we can plot that in this way:



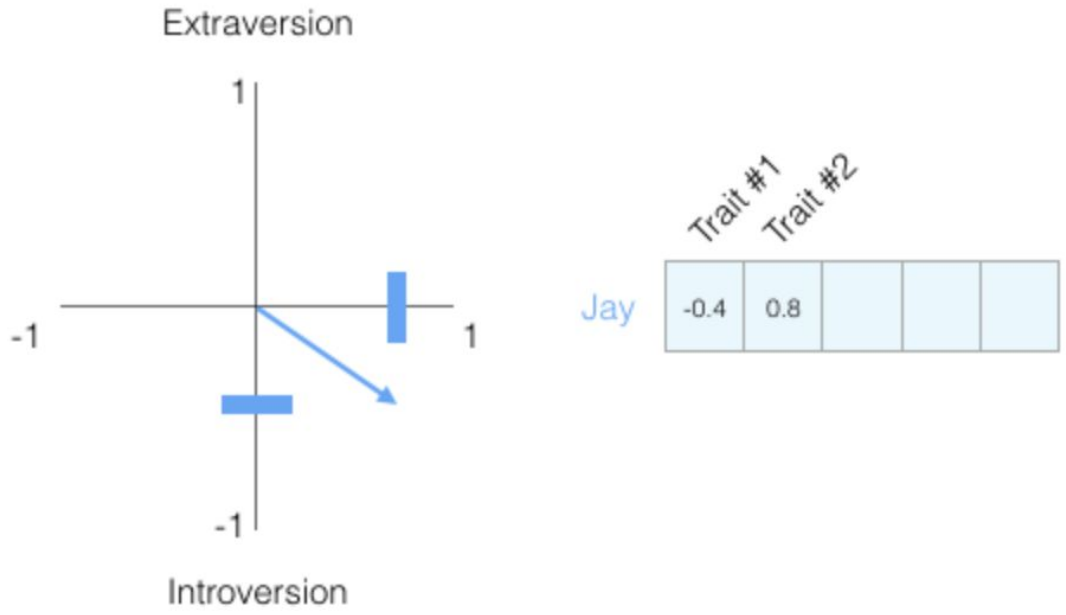


Let's switch the range to be from -1 to 1:





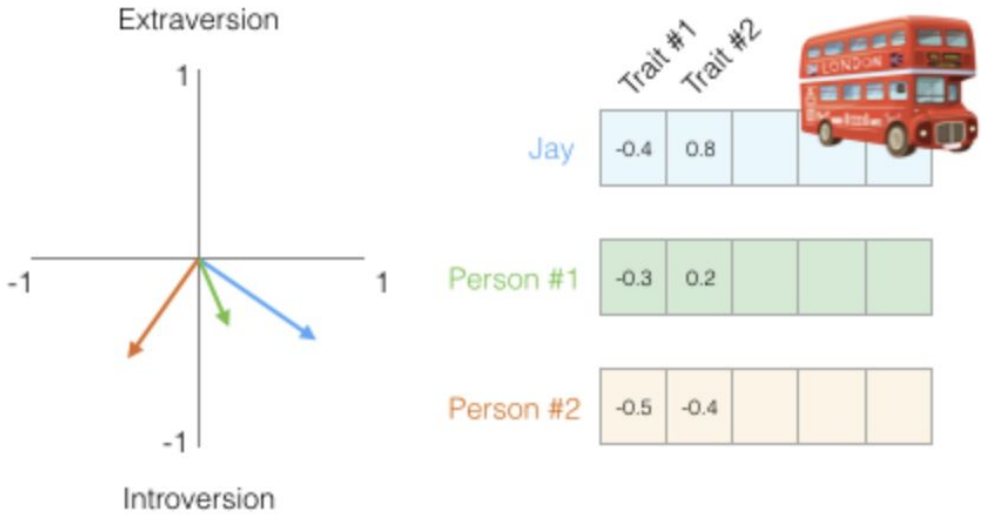
How well do you feel you know a person knowing only this one piece of information about them? Not much. People are complex. So let's add another dimension – the score of one other trait from the test.



We can represent the two dimensions as a point on the graph, or better yet, as a vector from the origin to that point. We have incredible tools to deal with vectors that will come in handy very shortly.

I've hidden which traits we're plotting just so you get used to not knowing what each dimension represents – but still getting a lot of value from the vector representation of a person's personality.

We can now say that this vector partially represents my personality. The usefulness of such representation comes when you want to compare two other people to me. Say I get hit by a `bus` and I need to be replaced by someone with a similar personality. In the following figure, which of the two people is more similar to me?



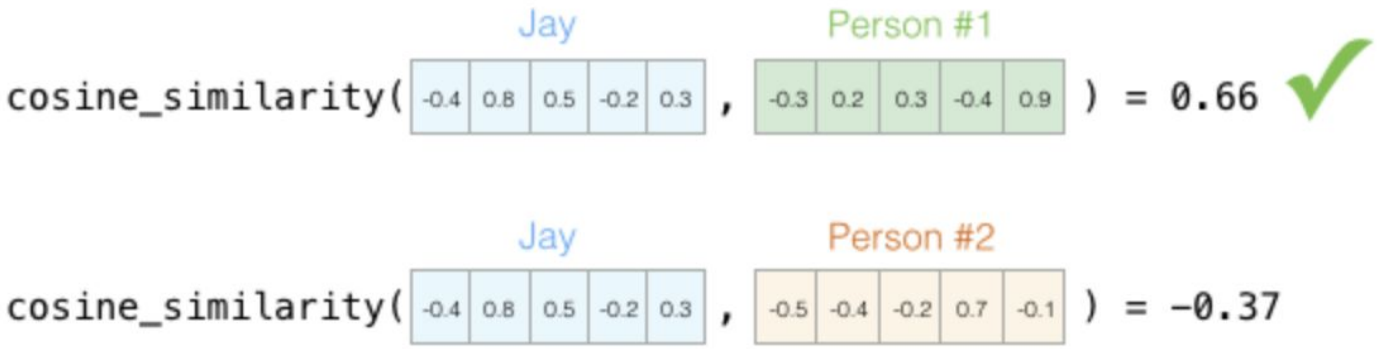
When dealing with vectors, a common way to calculate a similarity score is cosine_similarity:

Person #1 is more similar to me in personality. Vectors pointing at the same direction (length plays a role as well) have a higher cosine similarity score.

Yet again, two dimensions aren't enough to capture enough information about how different people are. Decades of psychology research have led to five major traits (and plenty of sub-traits). So let's use all five dimensions in our comparison:

	Trait #1	Trait #2	Trait #3	Trait #4	Trait #5
Jay	-0.4	0.8	0.5	-0.2	0.3
Person #1	-0.3	0.2	0.3	-0.4	0.9
Person #2	-0.5	-0.4	-0.2	0.7	-0.1

The problem with five dimensions is that we lose the ability to draw neat little arrows in two dimensions. This is a common challenge in machine learning where we often have to think in higher-dimensional space. The good thing is, though, that cosine_similarity still works. It works with any number of dimensions:



cosine_similarity works for any number of dimensions. These are much better scores because they're calculated based on a higher resolution representation of the things being compared.

At the end of this section, I want us to come out with two central ideas:

1. We can represent people (and things) as vectors of numbers (which is great for machines!).
2. We can easily calculate how similar vectors are to each other.

1- We can represent things (and people) as vectors of numbers
(Which is great for machines!)

Jay	-0.4	0.8	0.5	-0.2	0.3

2- We can easily calculate how similar vectors are to each other

The people most similar to Jay are:

	cosine_similarity ▼
Person #1	0.86
Person #2	0.5
Person #3	-0.20

Word Embeddings

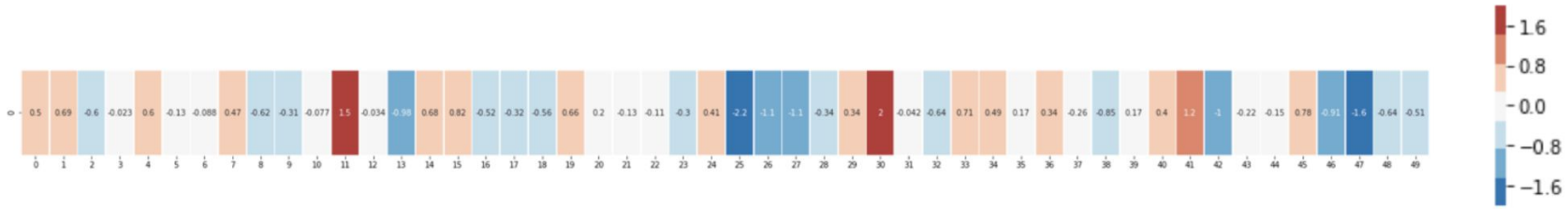
With this understanding, we can proceed to look at trained word-vector examples (also called word embeddings) and start looking at some of their interesting properties.

This is a word embedding for the word “king” (GloVe vector trained on Wikipedia):

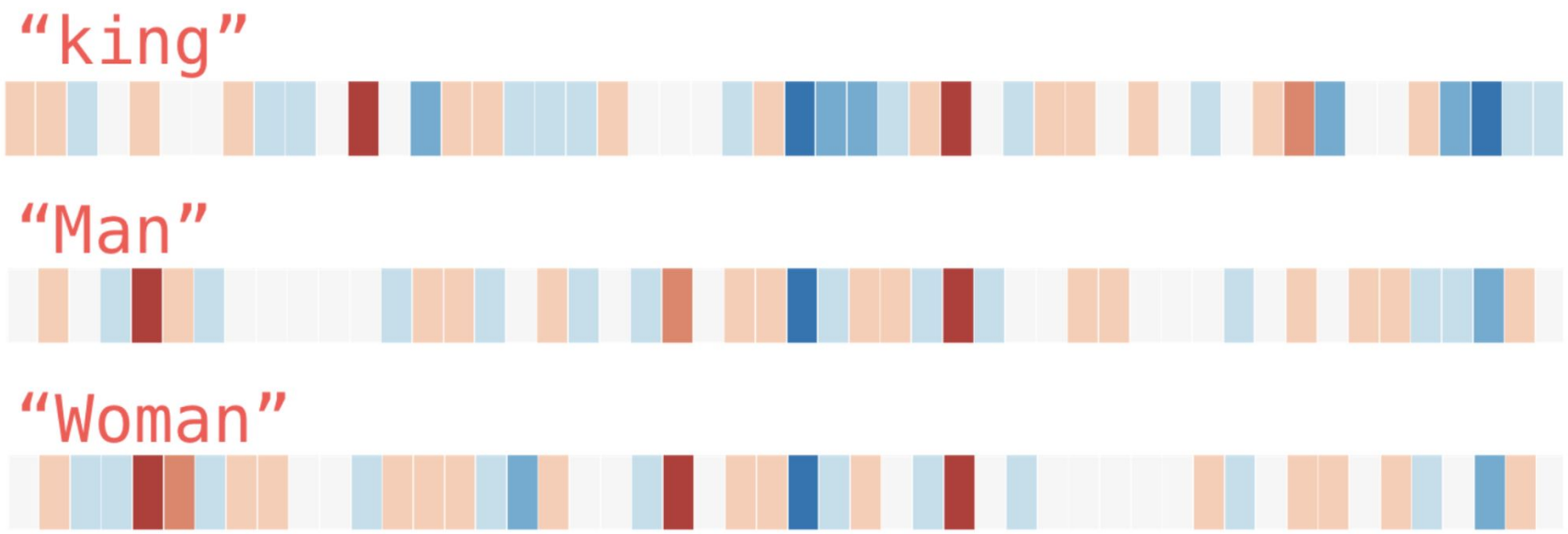
```
[ 0.50451 , 0.68607 , -0.59517 , -0.022801, 0.60046 , -0.13498 , -0.08813 , 0.47377 , -0.61798 , -0.31012 ,
-0.076666, 1.493 , -0.034189, -0.98173 , 0.68229 , 0.81722 , -0.51874 , -0.31503 , -0.55809 , 0.66421 , 0.1961
, -0.13495 , -0.11476 , -0.30344 , 0.41177 , -2.223 , -1.0756 , -1.0783 , -0.34354 , 0.33505 , 1.9927 ,
-0.04234 , -0.64319 , 0.71125 , 0.49159 , 0.16754 , 0.34344 , -0.25663 , -0.8523 , 0.1661 , 0.40102 , 1.1685 ,
-1.0137 , -0.21585 , -0.15155 , 0.78321 , -0.91241 , -1.6106 , -0.64426 , -0.51042 ]
```

It’s a list of 50 numbers. We can’t tell much by looking at the values. But let’s visualize it a bit so we can compare it other word vectors. Let’s put all these numbers in one row:

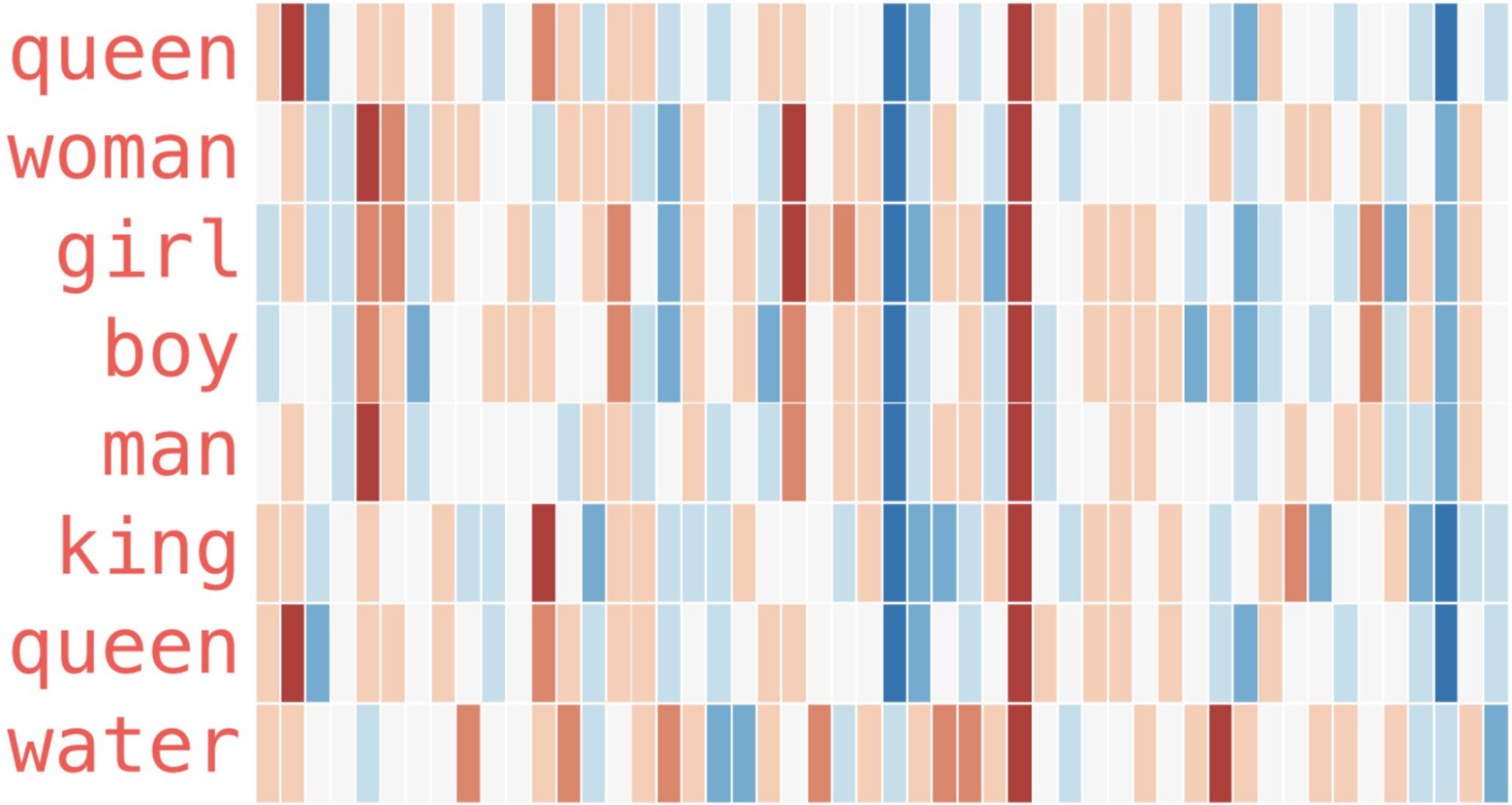
Let's color code the cells based on their values (red if they're close to 2, white if they're close to 0, blue if they're close to -2):



We'll proceed by ignoring the numbers and only looking at the colors to indicate the values of the cells. Let's now contrast "King" against other words:



Here's another list of examples (compare by vertically scanning the columns looking for columns with similar colors):



The famous examples that show an incredible property of embeddings is the concept of analogies. We can add and subtract word embeddings and arrive at interesting results. The most famous example is the formula: “king” - “man” + “woman”:

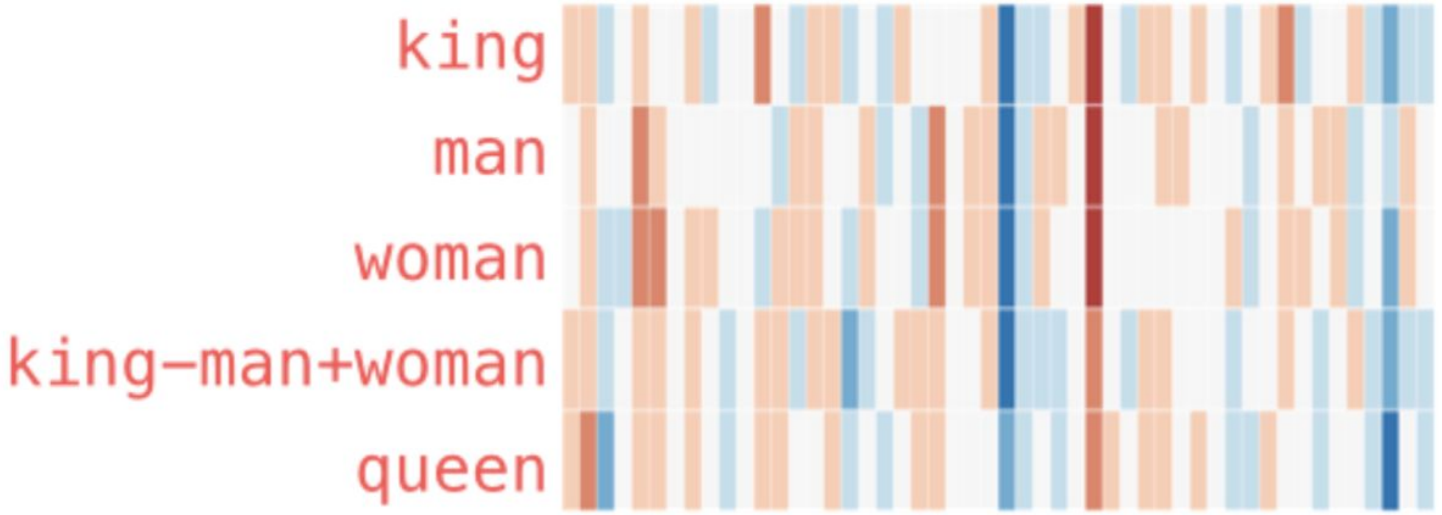
```
model.most_similar(positive=["king","woman"], negative=["man"])
```

```
[('queen', 0.8523603677749634),
 ('throne', 0.7664333581924438),
 ('prince', 0.7592144012451172),
 ('daughter', 0.7473883032798767),
 ('elizabeth', 0.7460219860076904),
 ('princess', 0.7424570322036743),
 ('kingdom', 0.7337411642074585),
 ('monarch', 0.721449077129364),
 ('eldest', 0.7184862494468689),
 ('widow', 0.7099430561065674)]
```

Using the Gensim library in python, we can add and subtract word vectors, and it would find the most similar words to the resulting vector. The image shows a list of the most similar words, each with its cosine similarity.

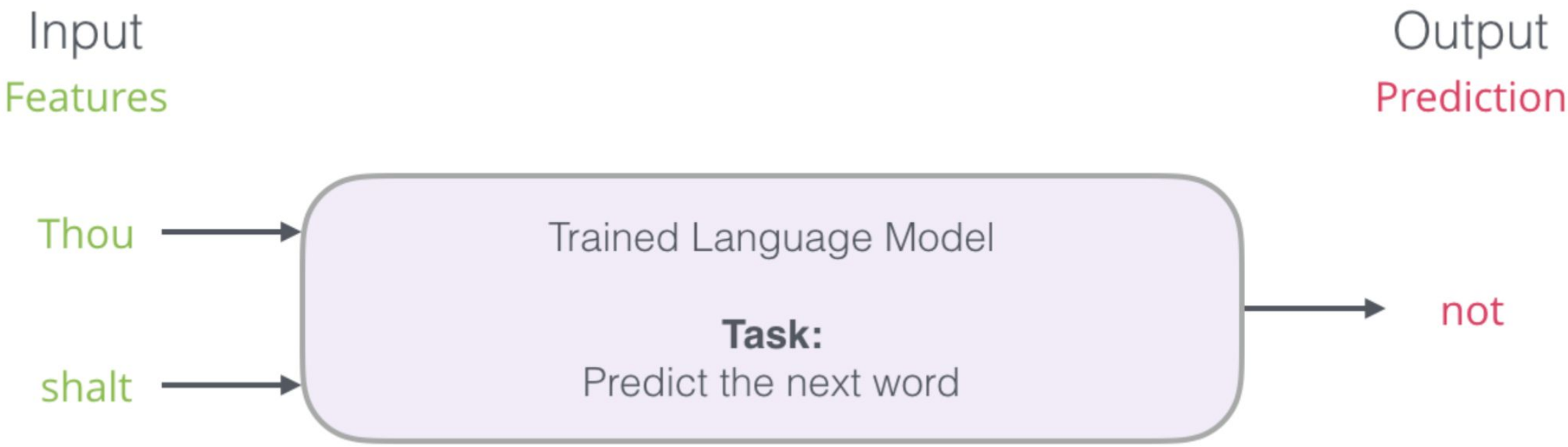
We can visualize this analogy as we did previously:

king - man + woman ~= queen

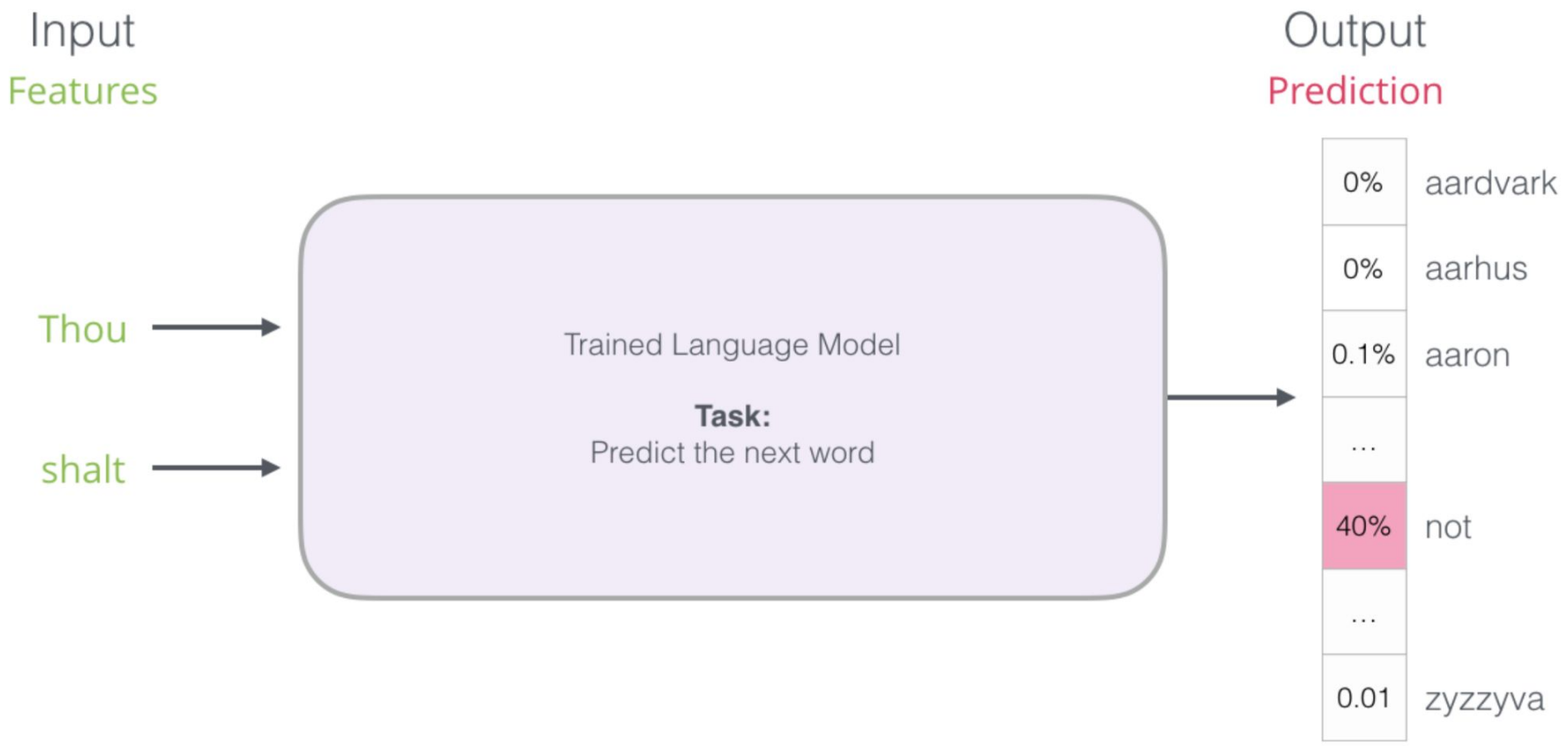


The resulting vector from "king-man+woman" doesn't exactly equal "queen", but "queen" is the closest word to it from the 400,000 word embeddings we have in this collection.

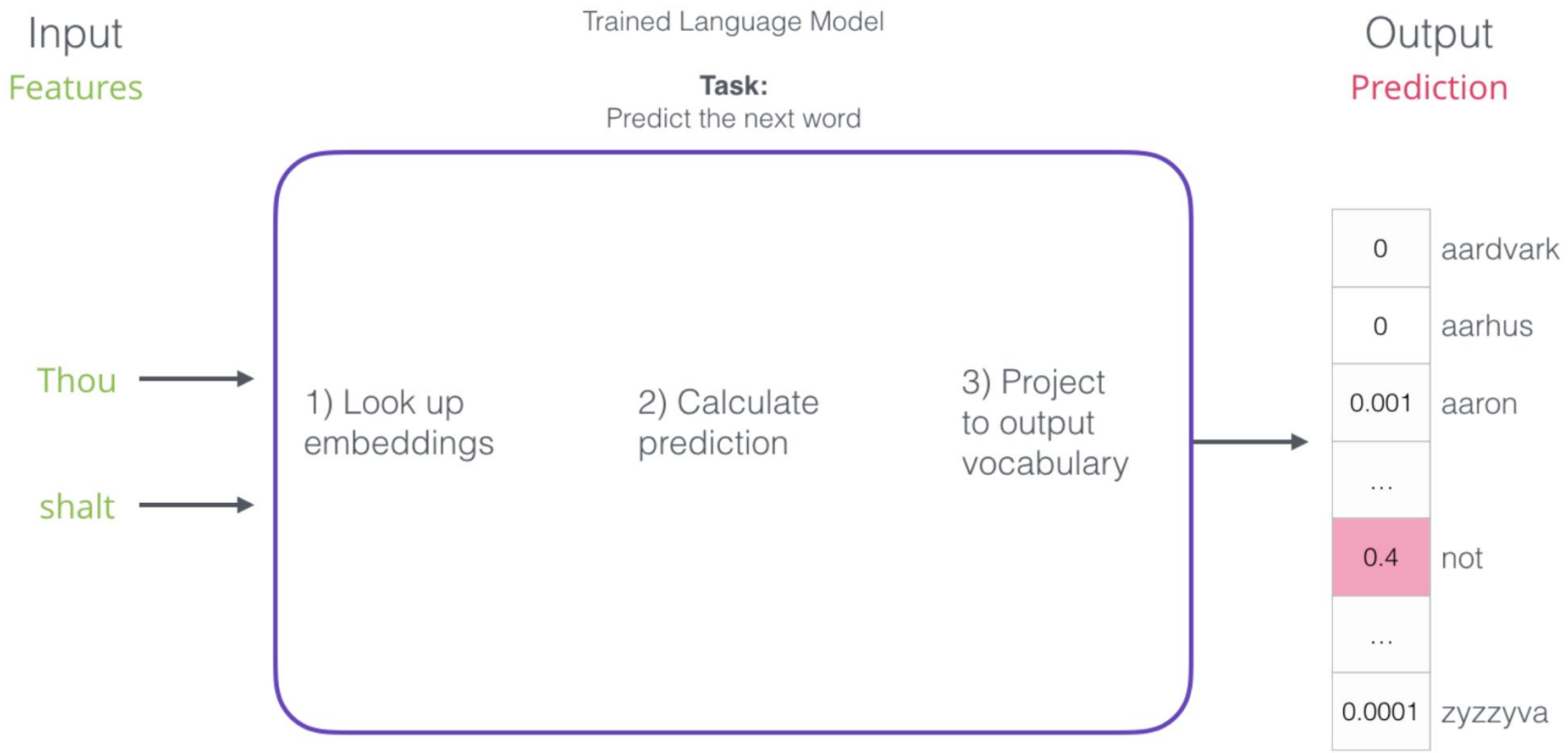
Language Model - NWP (Next Word Prediction)

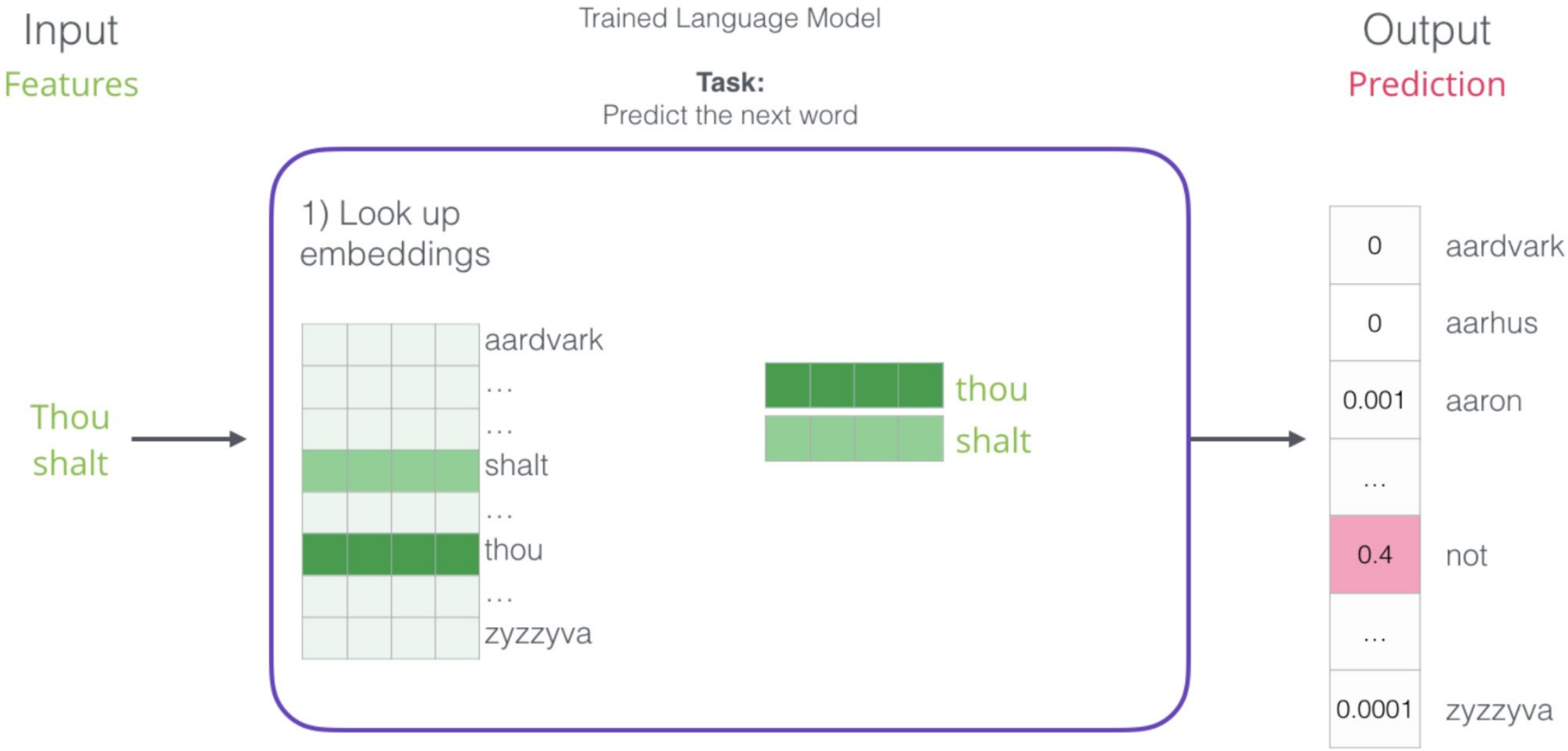


But in practice, the model doesn't output only one word. It actually outputs a probability score for all the words it knows (the model's "vocabulary", which can range from a few thousand to over a million words). The keyboard application then has to find the words with the highest scores, and present those to the user.



Three steps


Input
Features
Trained Language Model
Task:
Predict the next word
Output
Prediction
Thou
shalt
1) Look up embeddings
2) Calculate prediction
3) Project to output vocabulary
0
aardvark
0
aarhus
0.001
aaron
...
0.4
not
...
0.0001
zyzzyva


Input
Features
Trained Language Model
Task:
Predict the next word
Output
Prediction
Thou
shalt
1) Look up embeddings
aardvark
…
…
shalt
…
thou
…
zyzzyva
thou
shalt
0
aardvark
0
aarhus
0.001
aaron
…
0.4
not
…
0.0001
zyzzyva

Make the Dataset

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

CBOW

Instead of only looking two words before the target word, we can also look at two words after it.

Jay was hit by a _____ bus in...

by	a	red	bus	in

If we do this, the dataset we're virtually building and training the model against would look like this:

input 1	input 2	input 3	input 4	output
by	a	bus	in	red

Skipgram

The pink boxes are in different shades because this sliding window actually creates four separate samples in our training dataset:

input	output
red	by
red	a
red	bus
red	in

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	…

thou	shalt	not	make	a	machine	in	the	…

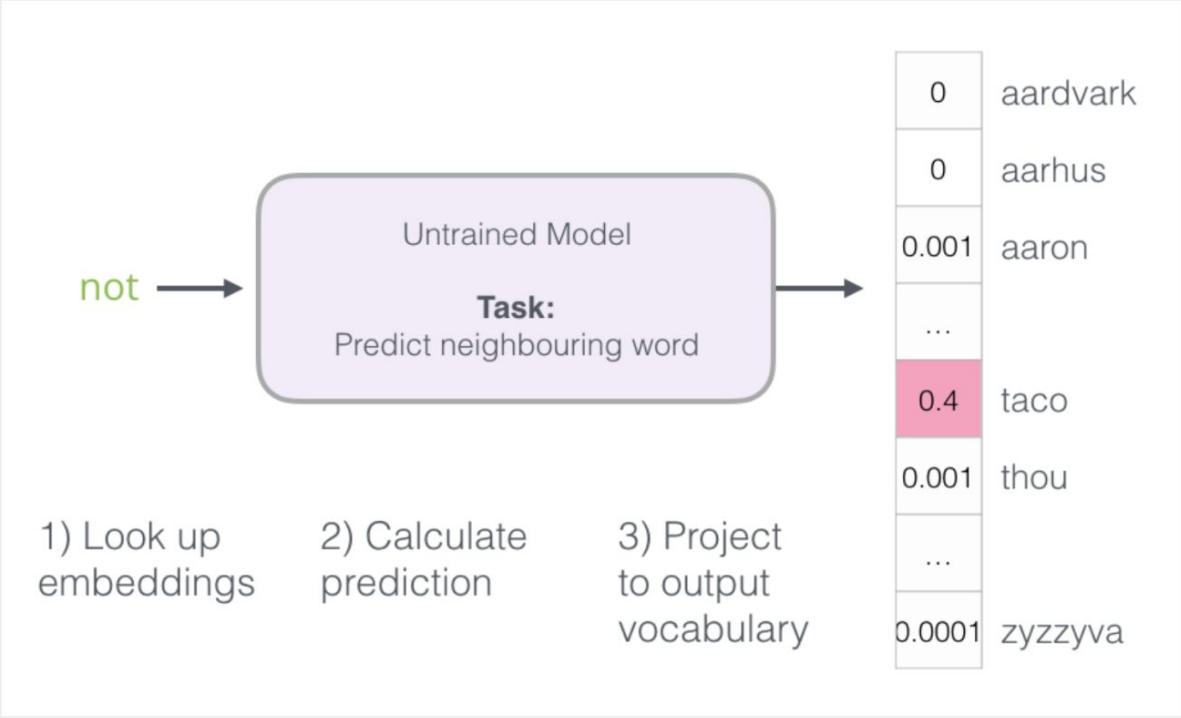
thou	shalt	not	make	a	machine	in	the	…

thou	shalt	not	make	a	machine	in	the	…

thou	shalt	not	make	a	machine	in	the	…

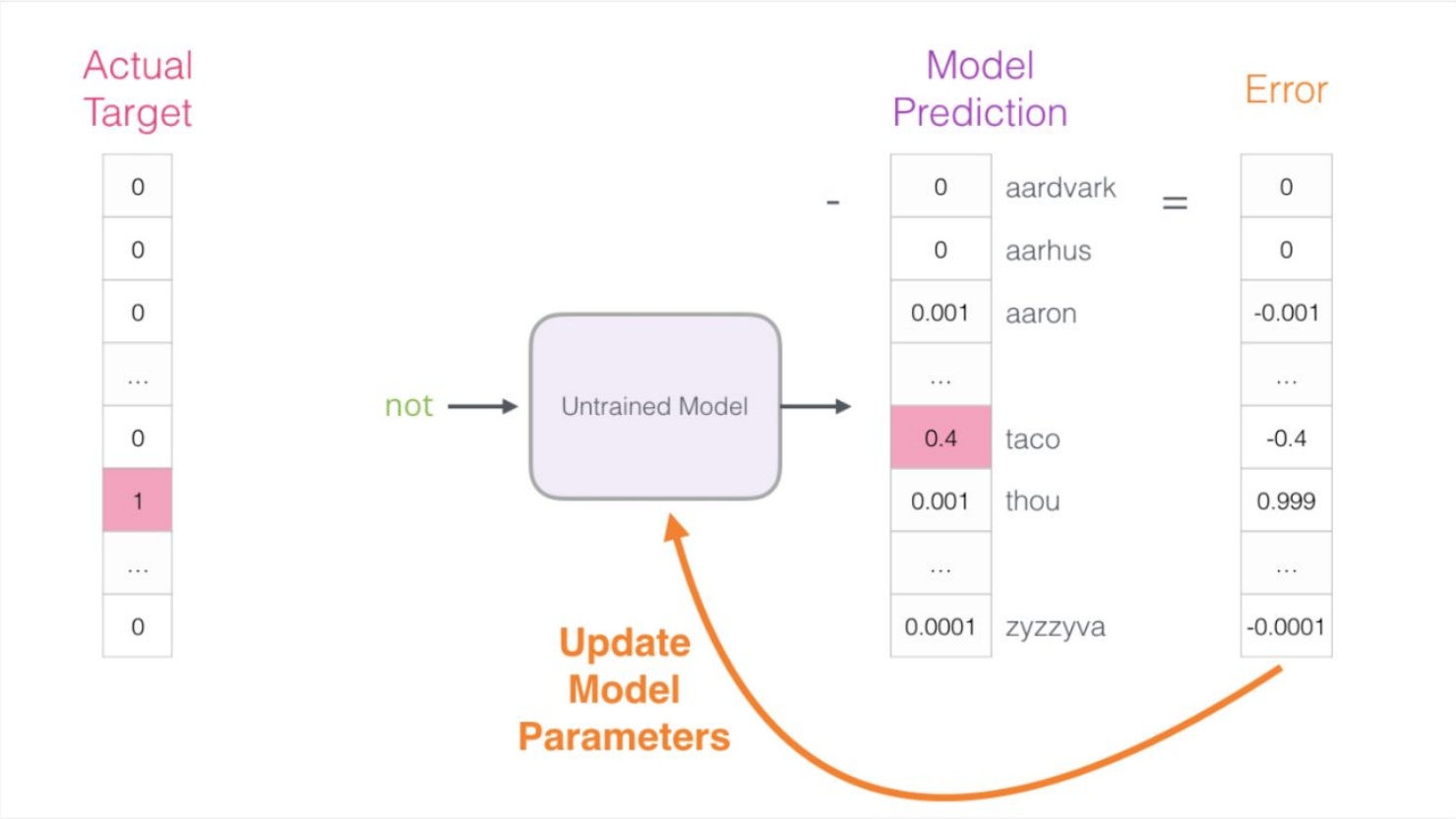
input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

We start with the first sample in our dataset. We grab the feature and feed to the untrained model asking it to predict an appropriate neighboring word.

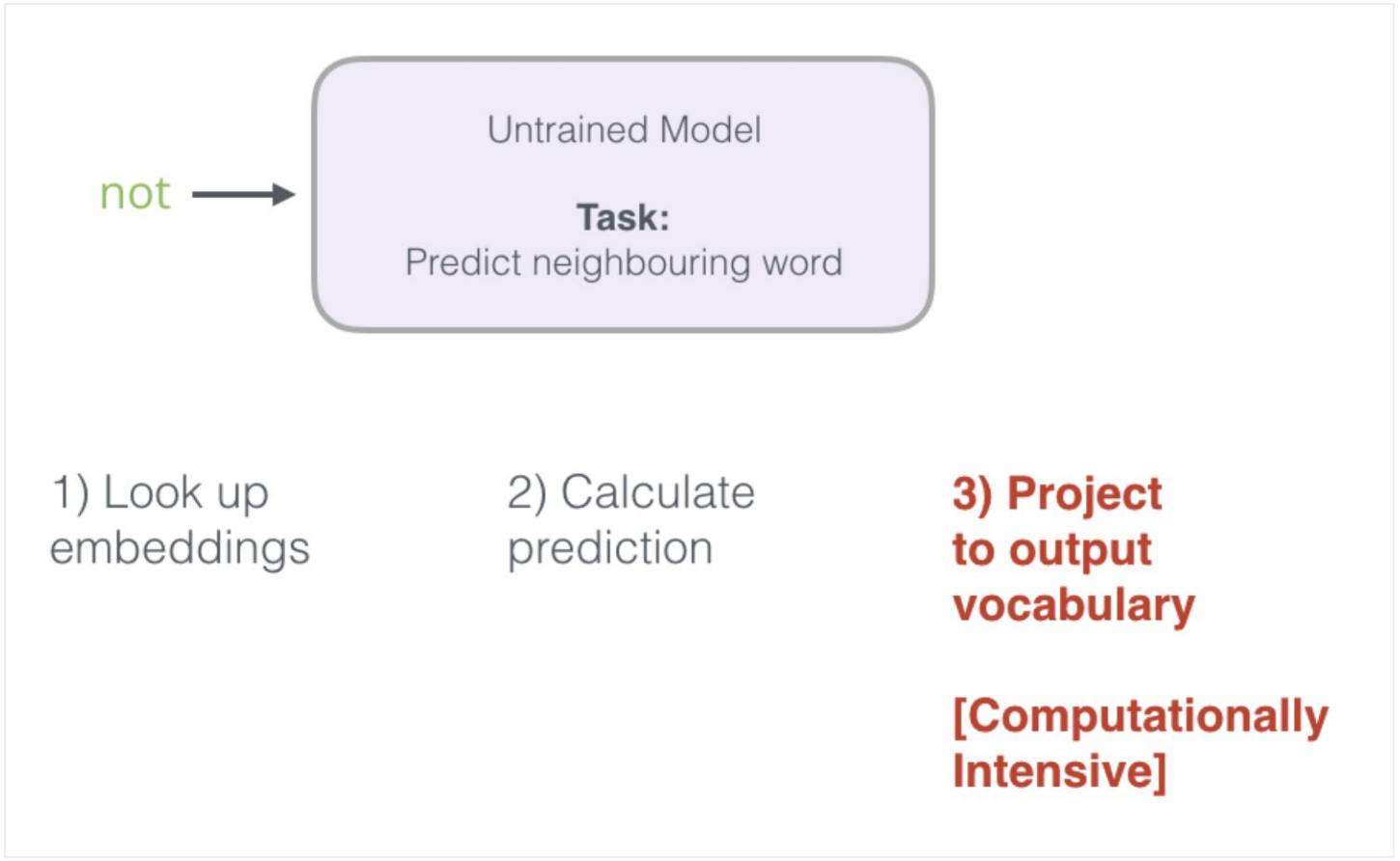


The model conducts the three steps and outputs a prediction vector (with a probability assigned to each word in its vocabulary). Since the model is untrained, it's prediction is sure to be wrong at this stage. But that's okay. We know what word it should have guessed – the label/output cell in the row we're currently using to train the model:

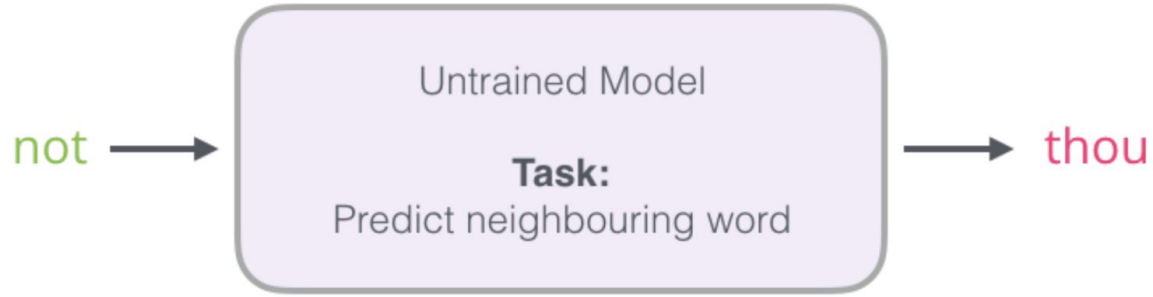
This error vector can now be used to update the model so the next time, it's a little more likely to guess `thou` when it gets `not` as input.



Changing our approach

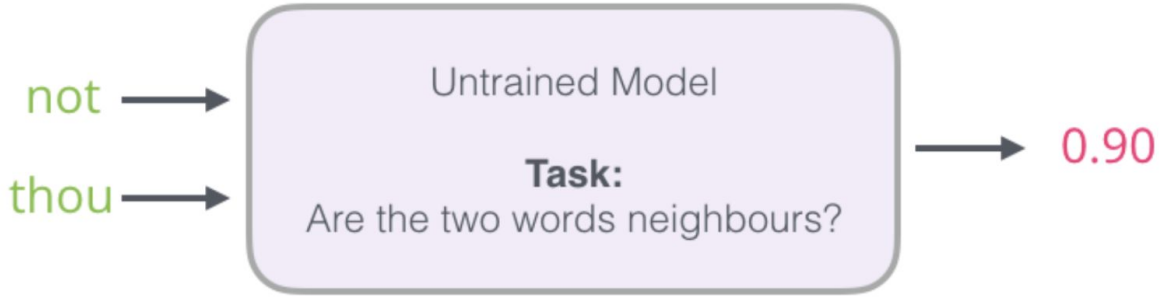


Change Task from



And switch it to a model that takes the input and output word, and outputs a score indicating if they're neighbors or not (0 for "not neighbors", 1 for "neighbors").

To:



This simple switch changes the model we need from a neural network, to a logistic regression model – thus it becomes much simpler and much faster to calculate.

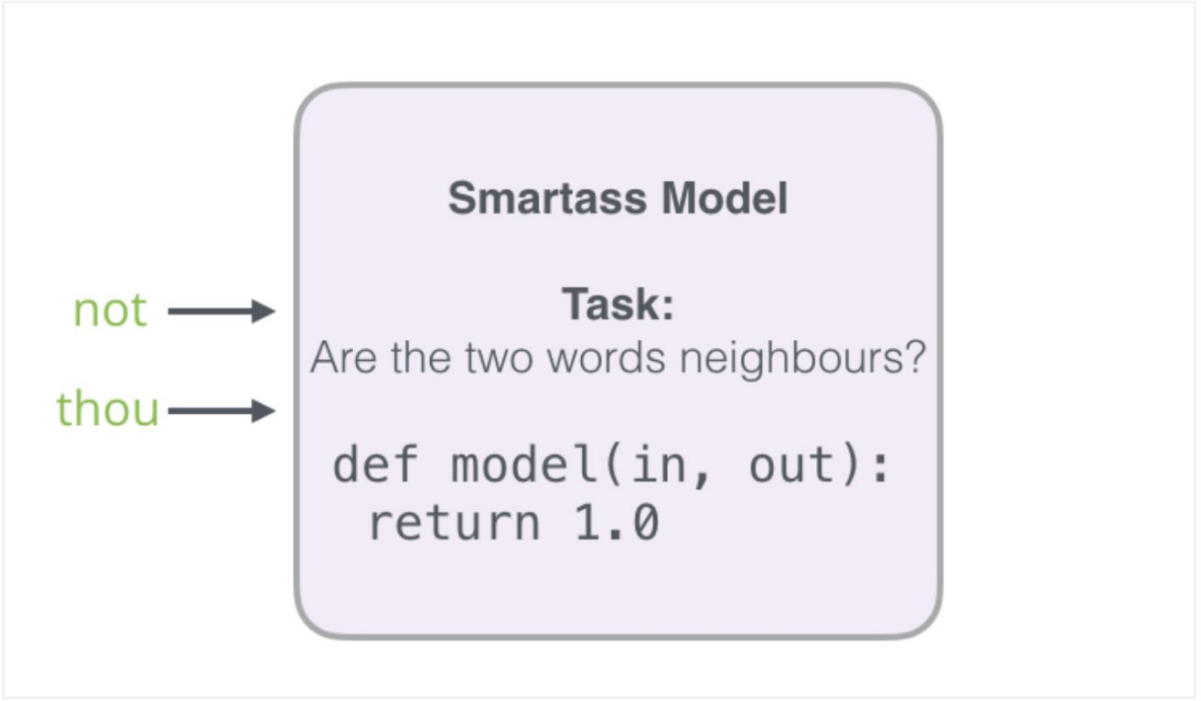
This switch requires we switch the structure of our dataset – the label is now a new column with values 0 or 1. They will be all 1 since all the words we added are neighbors.

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

Negative Sampling

This can now be computed at blazing speed – processing millions of examples in minutes. But there's one loophole we need to close. If all of our examples are positive (target: 1), we open ourself to the possibility of a smartass model that always returns 1 – achieving 100% accuracy, but learning nothing and generating garbage embeddings.



Pick randomly from vocabulary (random sampling)

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

Word	Count	Probability
aardvark		
aarhus		
aaron		
taco		
thou		
zyzzyva		

Skipgram with Negative Sampling (SGNS)

We have now covered two of the central ideas in word2vec: as a pair, they're called skipgram with negative sampling.

Skipgram

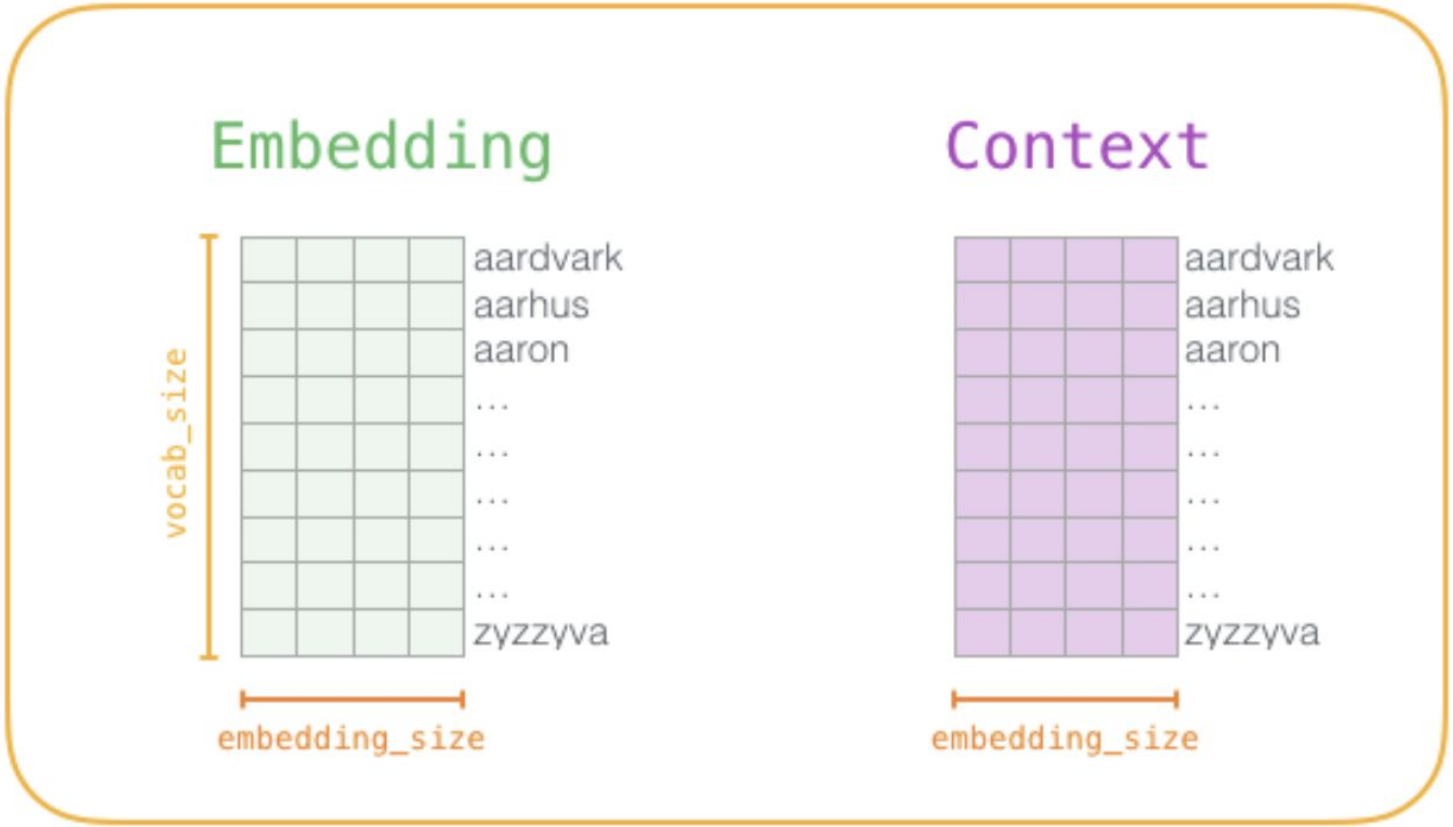
shalt	not	make	a	machine

input	output
make	shalt
make	not
make	a
make	machine

Negative Sampling

input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0

Word2vec Training Process

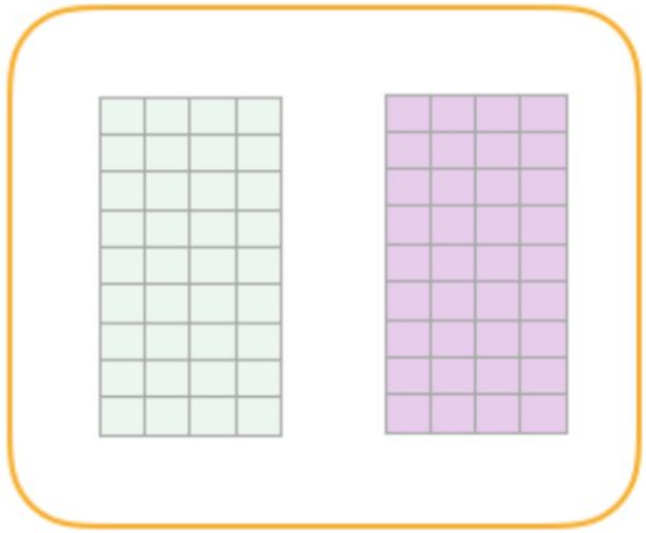


At the start of the training process, we initialize these matrices with random values. Then we start the training process. In each training step, we take one positive example and its associated negative examples. Let's take our first group:

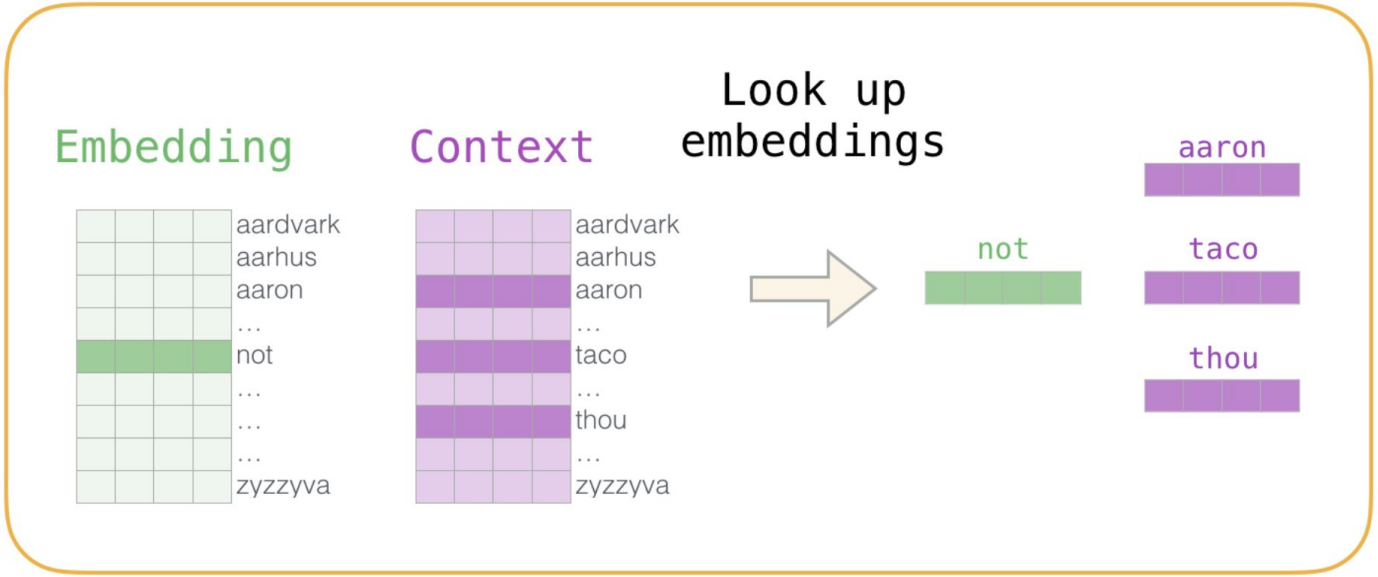
dataset

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...

model



Now we have four words: the input word `not` and output/context words: `thou` (the actual neighbor), `aaron`, and `taco` (the negative examples). We proceed to look up their embeddings – for the input word, we look in the `Embedding` matrix. For the context words, we look in the `Context` matrix (even though both matrices have an embedding for every word in our vocabulary).



Then, we take the dot product of the input embedding with each of the context embeddings. In each case, that would result in a number, that number indicates the similarity of the input and context embeddings

input word	output word	target	input • output
not	thou	1	0.2
not	aaron	0	-1.11
not	taco	0	0.74

Then, we take the dot product of the input embedding with each of the context embeddings. In each case, that would result in a number, that number indicates the similarity of the input and context embeddings

input word	output word	target	input • output
not	thou	1	0.2
not	aaron	0	-1.11
not	taco	0	0.74

Now we need a way to turn these scores into something that looks like probabilities – we need them to all be positive and have values between zero and one. This is a great task for sigmoid, the logistic operation.

input word	output word	target	input • output	sigmoid()
not	thou	1	0.2	0.55
not	aaron	0	-1.11	0.25
not	taco	0	0.74	0.68

1. Hyperparameters
   a. Window size
   b. Number of Neg. samples

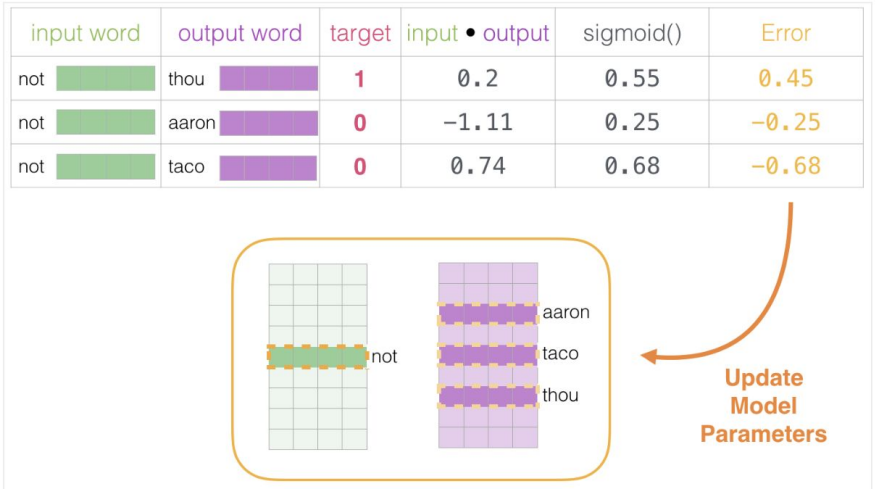
And we can now treat the output of the sigmoid operations as the model's output for these examples. You can see that `taco` has the highest score and `aaron` still has the lowest score both before and after the sigmoid operations.

Now that the untrained model has made a prediction, and seeing as though we have an actual target label to compare against, let's calculate how much error is in the model's prediction. To do that, we just subtract the sigmoid scores from the target labels.

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68

```
error = target - sigmoid_scores
```

Here comes the "learning" part of "machine learning". We can now use this error score to adjust the embeddings of `not`, `thou`, `aaron`, and `taco` so that the next time we make this calculation, the result would be closer to the target scores.



References

1. https://jalammar.github.io/illustrated-word2vec/