

# Word Embeddings

Dr M Janaki Meena

# Word Embeddings

- Word embedding is one of the most fascinating ideas in machine learning
- Siri, Google Assistant, Alexa, Google Translate, or even smartphone keyboard with next-word prediction is modelled using word embeddings
- quite a development over the last couple of decades in using embeddings for neural models

# Word2vec

- Since 2013 this technique is used to create word embeddings
- Some of its concepts have been shown to be effective in
  - creating recommendation engines
  - Making sense of sequential data even in commercial, non-language tasks
- Companies like Airbnb, Alibaba, Spotify, and Anghami have all benefitted from carving out this brilliant piece of machinery from the world of NLP
- They have used this technique in production to empower a new breed of recommendation engines

# Personality Embeddings: What are you like?

- Represent a person by his characteristics
- Characteristics may include - height, weight, gender, coding skills, communication skills, research potential, creativity etc

# Big Five personality traits

- these are tests that ask you a list of questions
- Can really tell you a lot about yourself and is shown to have predictive ability in academic, personal, and professional success
- then score you on a number of axes
- a list of questions

# Example List of Questions

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

# Plot the Scores

- Imagine I've scored 38/100 as my introversion/extraversion score

Extraversion

100

0

Introversion

Jay

Extraversion

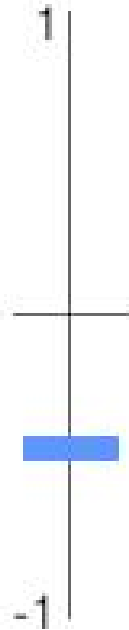
38



# Plot the Scores

- Switch the range to be from -1 to 1:

Extraversion



Introversion

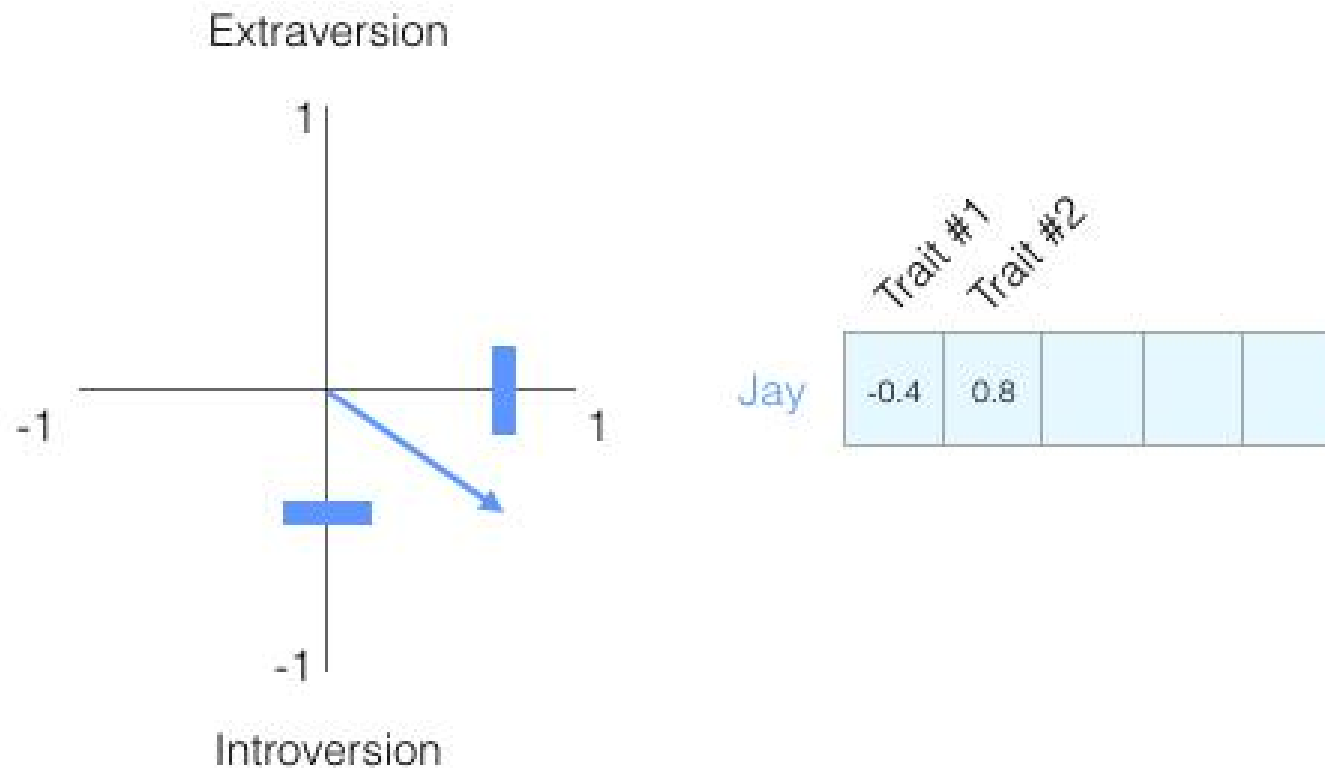
Jay





# Plot the Scores

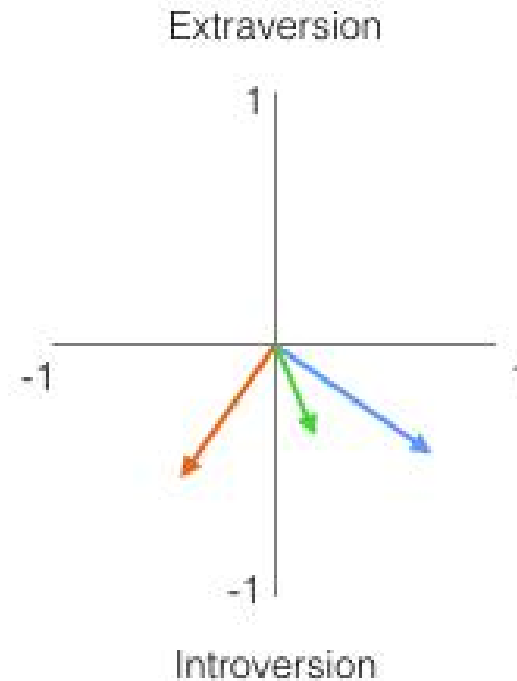
- Knowing only this one piece of information about them we may not know much about a person




this vector partially represents my personality

# Usefulness of Such Representation

- 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?



	Trait #1	Trait #2			
Jay	-0.4	0.8			
Person #1	-0.3	0.2			
Person #2	-0.5	-0.4			



# Similarity Between Two Vectors

- a common way to calculate a similarity score between two vectors is by cosine\_similarity:

$$\text{Cos}(x, y) = x \cdot y / ||x|| * ||y||$$

where,

$x \cdot y$  = product (dot) of the vectors 'x' and 'y'.

$||x||$  and  $||y||$  = length of the two vectors 'x' and 'y'.

$||x|| * ||y||$  = cross product of the two vectors 'x' and 'y'.

# Example

Find Cosine similarity between two vectors – 'x' and 'y'

$$x = \{ 3, 2, 0, 5 \}$$

$$y = \{ 1, 0, 0, 0 \}$$

$$\text{Formula - } \text{Cos}(x, y) = x \cdot y / \|x\| * \|y\|$$

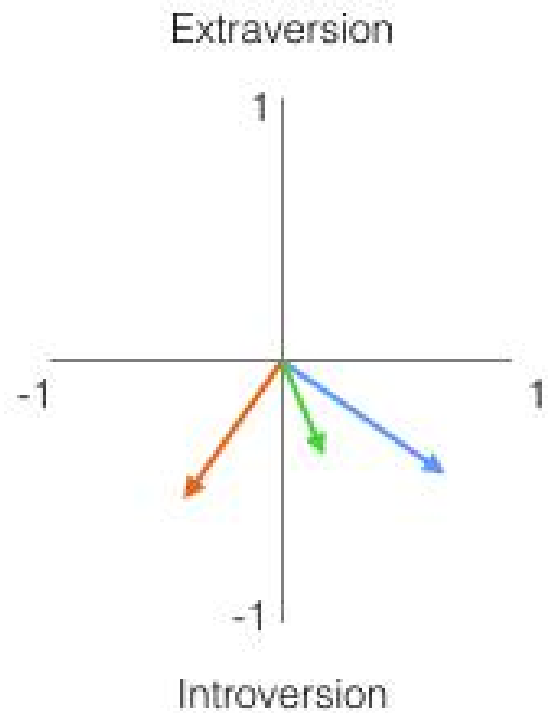
$$x \cdot y = 3*1 + 2*0 + 0*0 + 5*0 = 3$$

$$\|x\| = \sqrt{(3)^2 + (2)^2 + (0)^2 + (5)^2} = 6.16$$

$$\|y\| = \sqrt{(1)^2 + (0)^2 + (0)^2 + (0)^2} = 1$$

$$\therefore \text{Cos}(x, y) = 3 / (6.16 * 1) = 0.49$$

# Apply for Our Case to Find a Person Similar to Jay



	Trait #1	Trait #2	
Jay	-0.4	0.8	

Person #1	-0.3	0.2			
-----------	------	-----	--	--	--

Person #2	-0.5	-0.4			
-----------	------	------	--	--	--

$$\text{cosine\_similarity}\left(\begin{array}{|c|c|} \hline \text{Jay} \\ \hline -0.4 & 0.8 \\ \hline \end{array}, \begin{array}{|c|c|} \hline \text{Person \#1} \\ \hline -0.3 & 0.2 \\ \hline \end{array}\right) = 0.87$$



$$\text{cosine\_similarity}\left(\begin{array}{|c|c|} \hline \text{Jay} \\ \hline -0.4 & 0.8 \\ \hline \end{array}, \begin{array}{|c|c|} \hline \text{Person \#2} \\ \hline -0.5 & -0.4 \\ \hline \end{array}\right) = -0.20$$

# More Dimensions

- 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

# More Dimensions

- 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

# Higher-Dimensional Space

- We lose the ability to draw neat little arrows as in two dimensions
- Common challenge in machine learning where we often have to think in higher-dimensional space
- Good thing is, though, that cosine\_similarity still works. It works with any number of dimensions

$$\text{cosine\_similarity}(\text{Jay}, \text{Person \#1}) = 0.66 \quad \checkmark$$

-0.4	0.8	0.5	-0.2	0.3
------	-----	-----	------	-----

-0.3	0.2	0.3	-0.4	0.9
------	-----	-----	------	-----

$$\text{cosine\_similarity}(\text{Jay}, \text{Person \#2}) = -0.37$$

-0.4	0.8	0.5	-0.2	0.3
------	-----	-----	------	-----

-0.5	-0.4	-0.2	0.7	-0.1
------	------	------	-----	------



# Two Central Ideas

- We can represent people (and things) as vectors of numbers
- 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

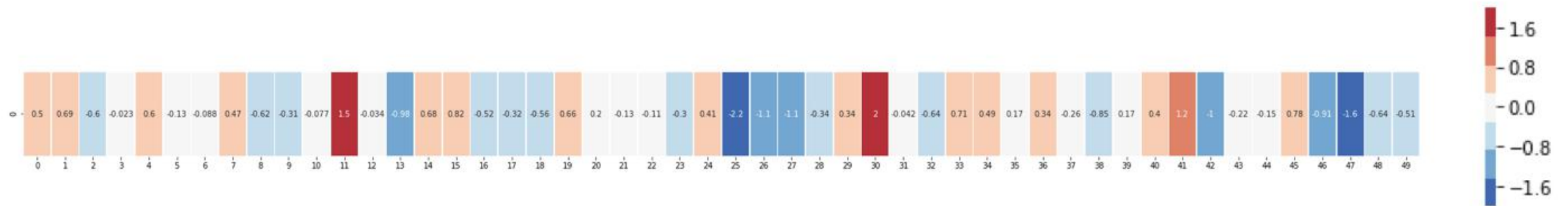
# Word Embeddings

- Look at trained word-vector examples (also called word embeddings) and start looking at some of their interesting properties
- Word embedding for the word “king” (GloVe vector trained on Wikipedia):
- It’s a vector of size 50 numbers
- There are also vectors of size 100, 200 and 300

```
[ 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 ]
```

# Visualize Word Embeddings

- Not easy interpret as numbers
- Let's visualize it a bit so we can compare it other word vectors
- 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):



# Contrast “King” against other words

“king”

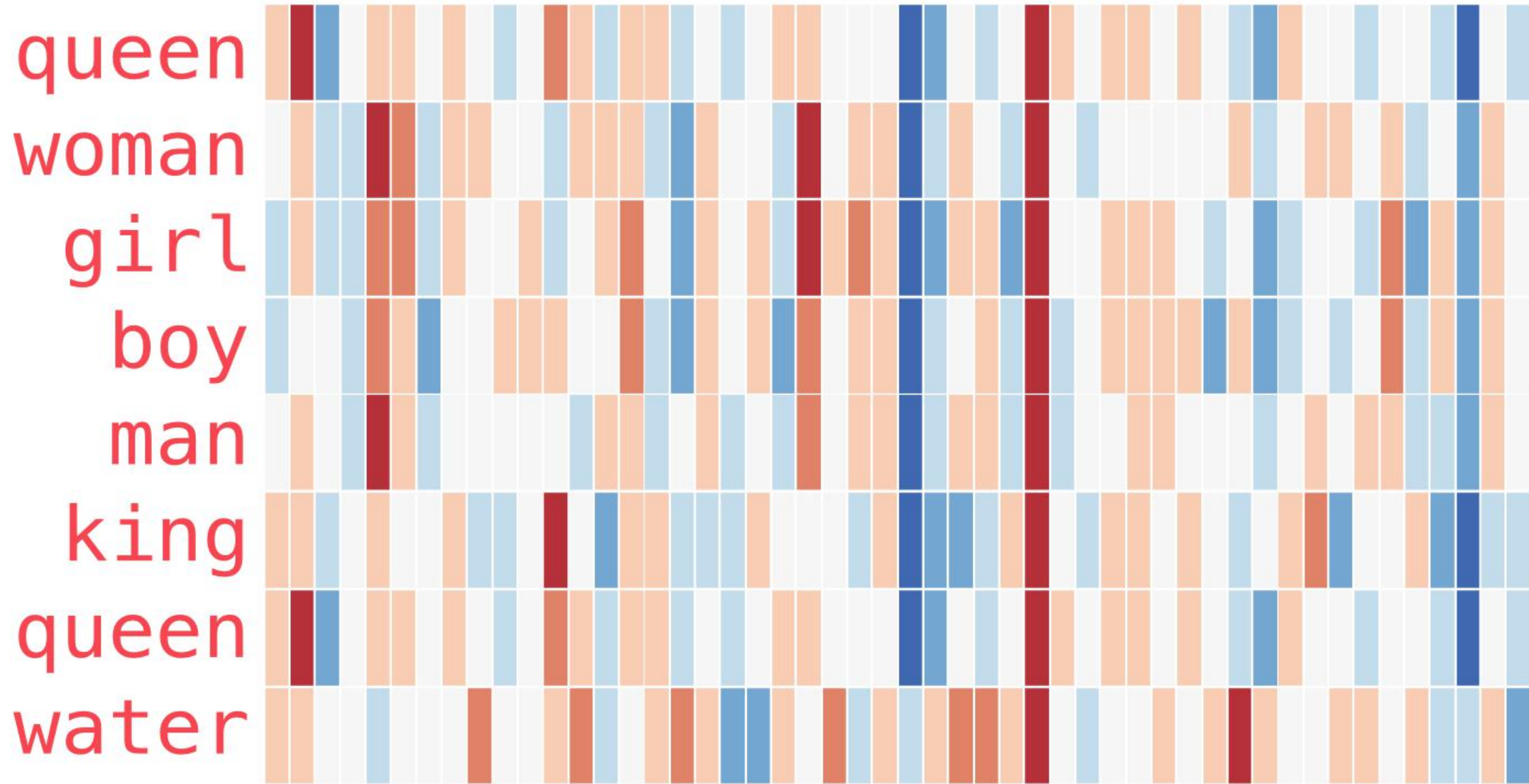


“Man”

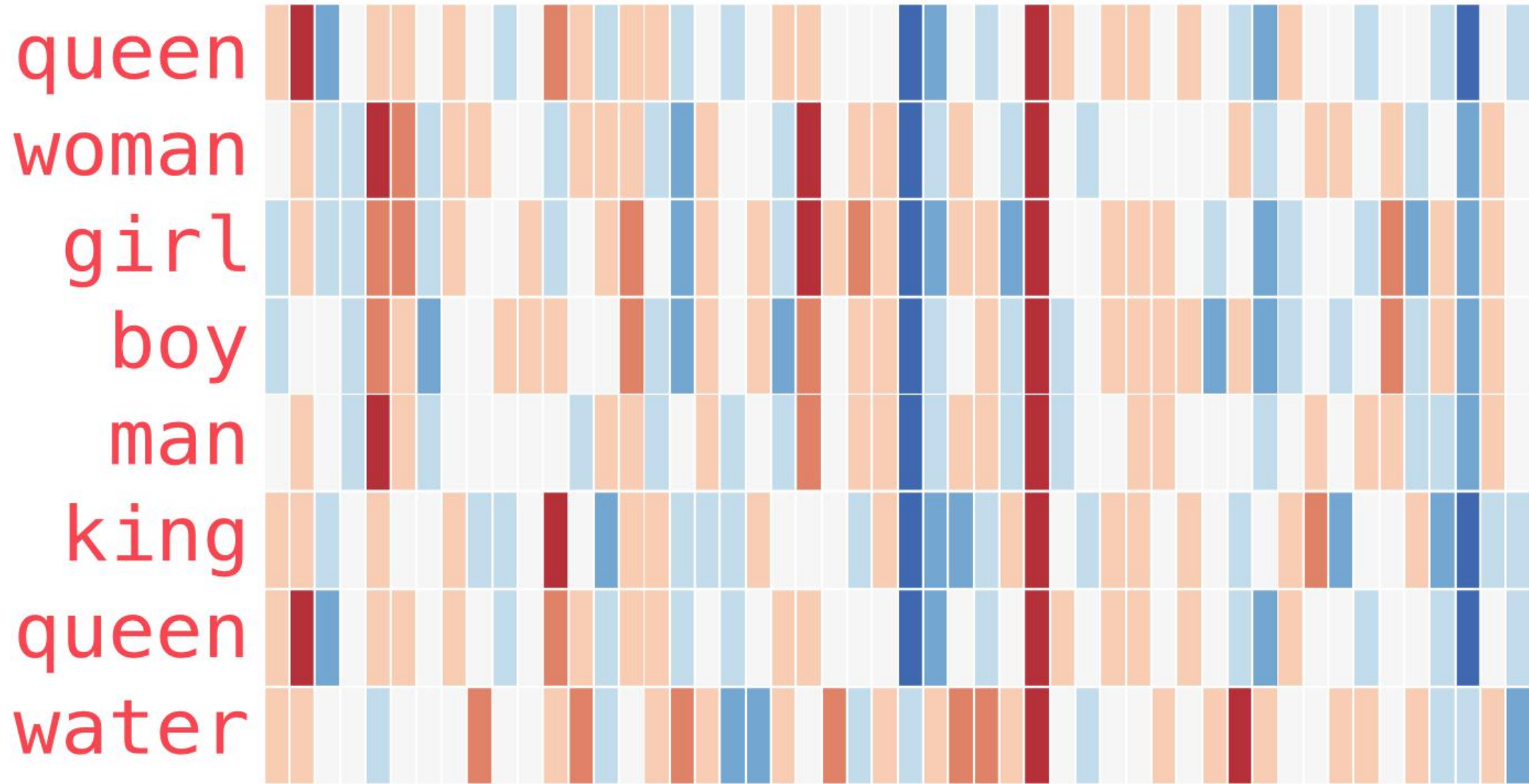


“Woman”



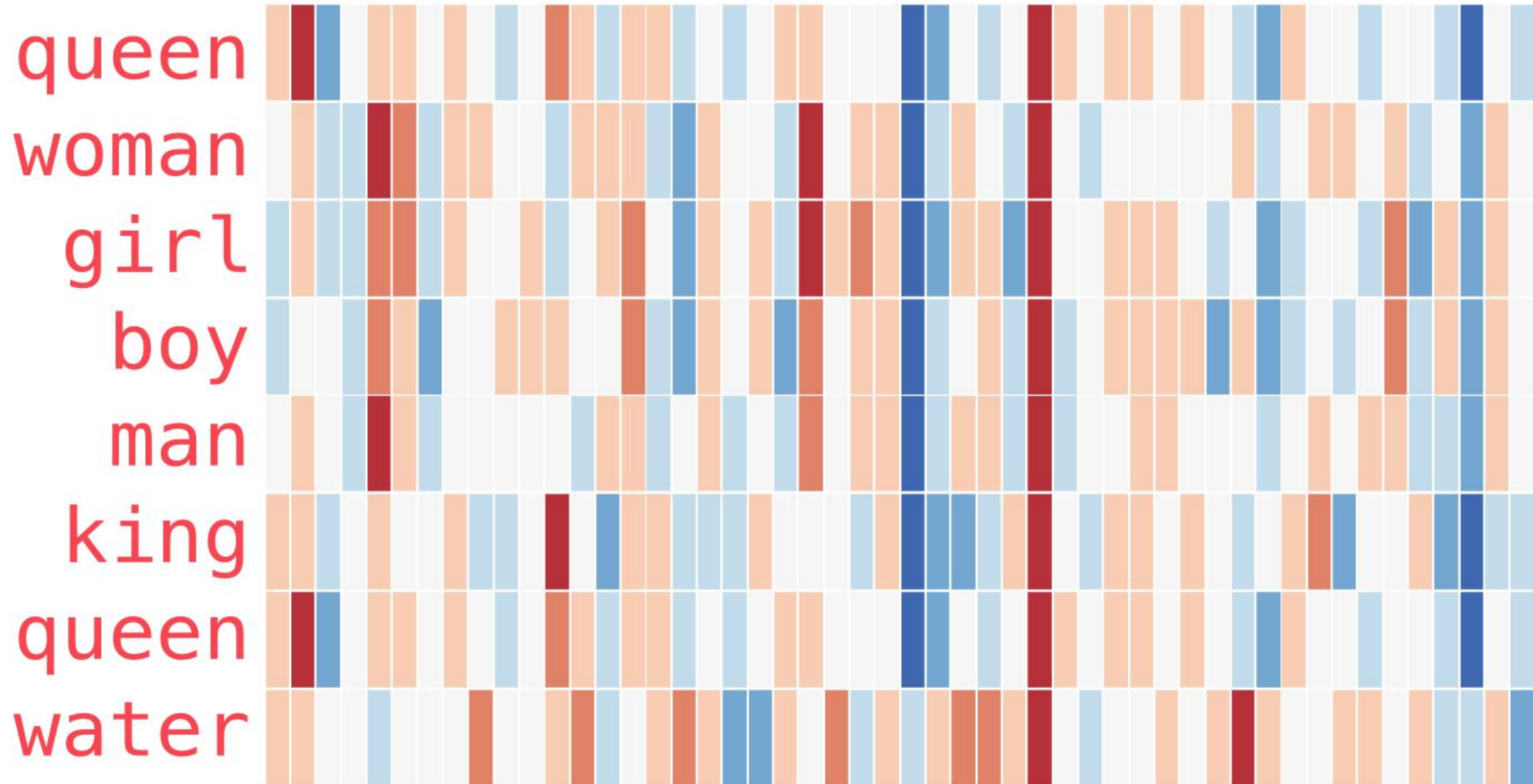


- A straight red column through all of these different words
- They're similar along that dimension (and we don't know what each dimensions codes for)

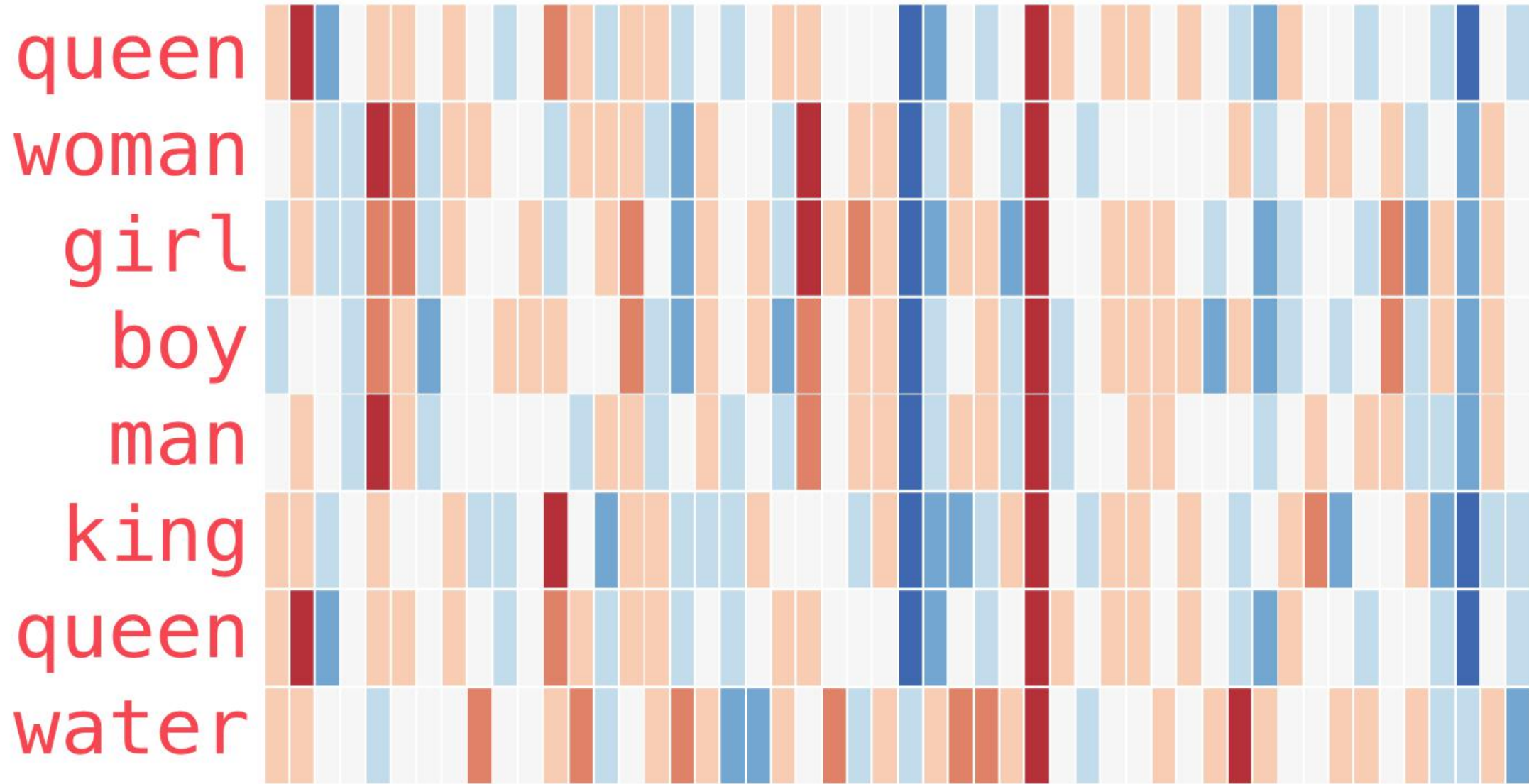


- You can see how “woman” and “girl” are similar to each other in a lot of places



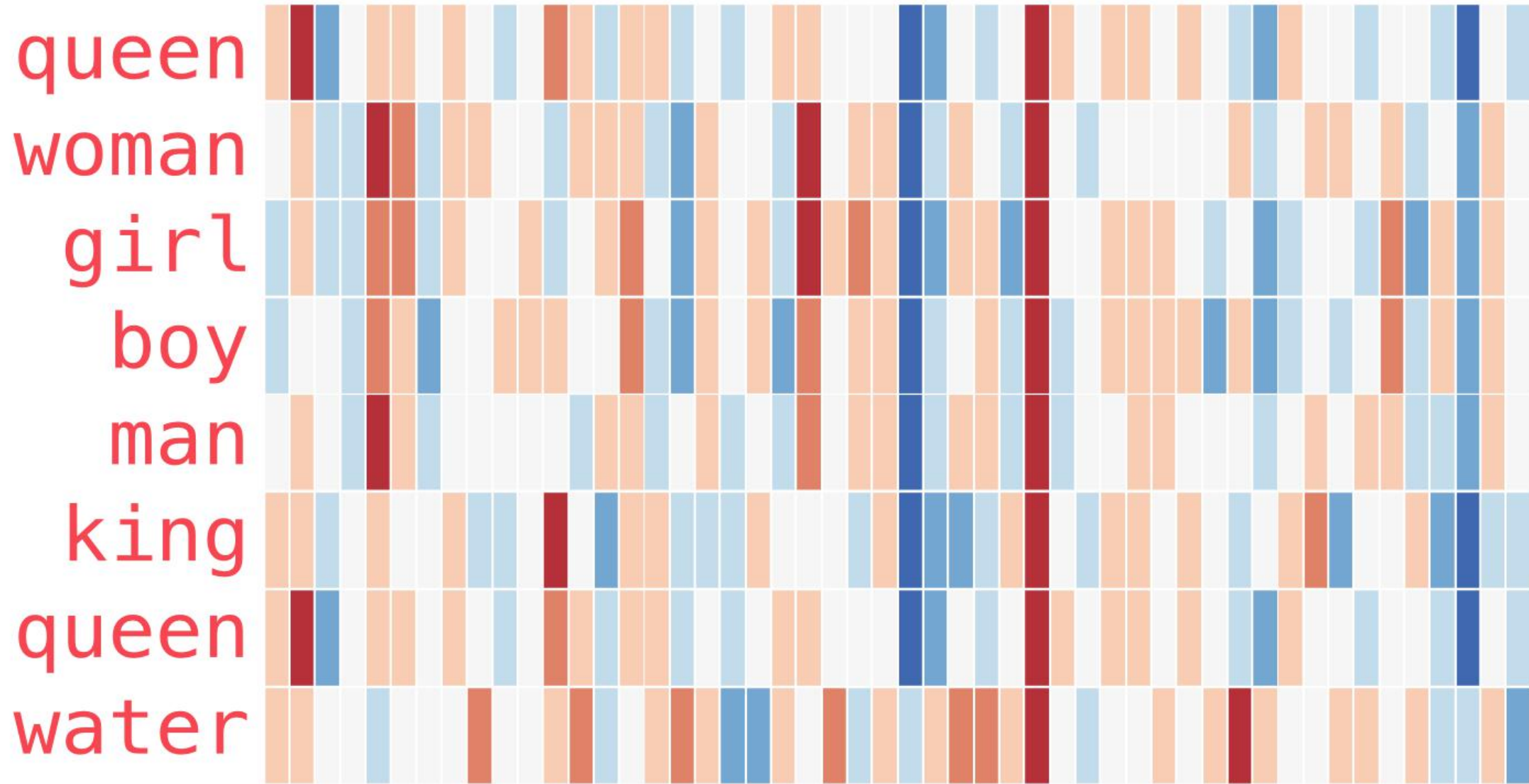


- The same with “man” and “boy”

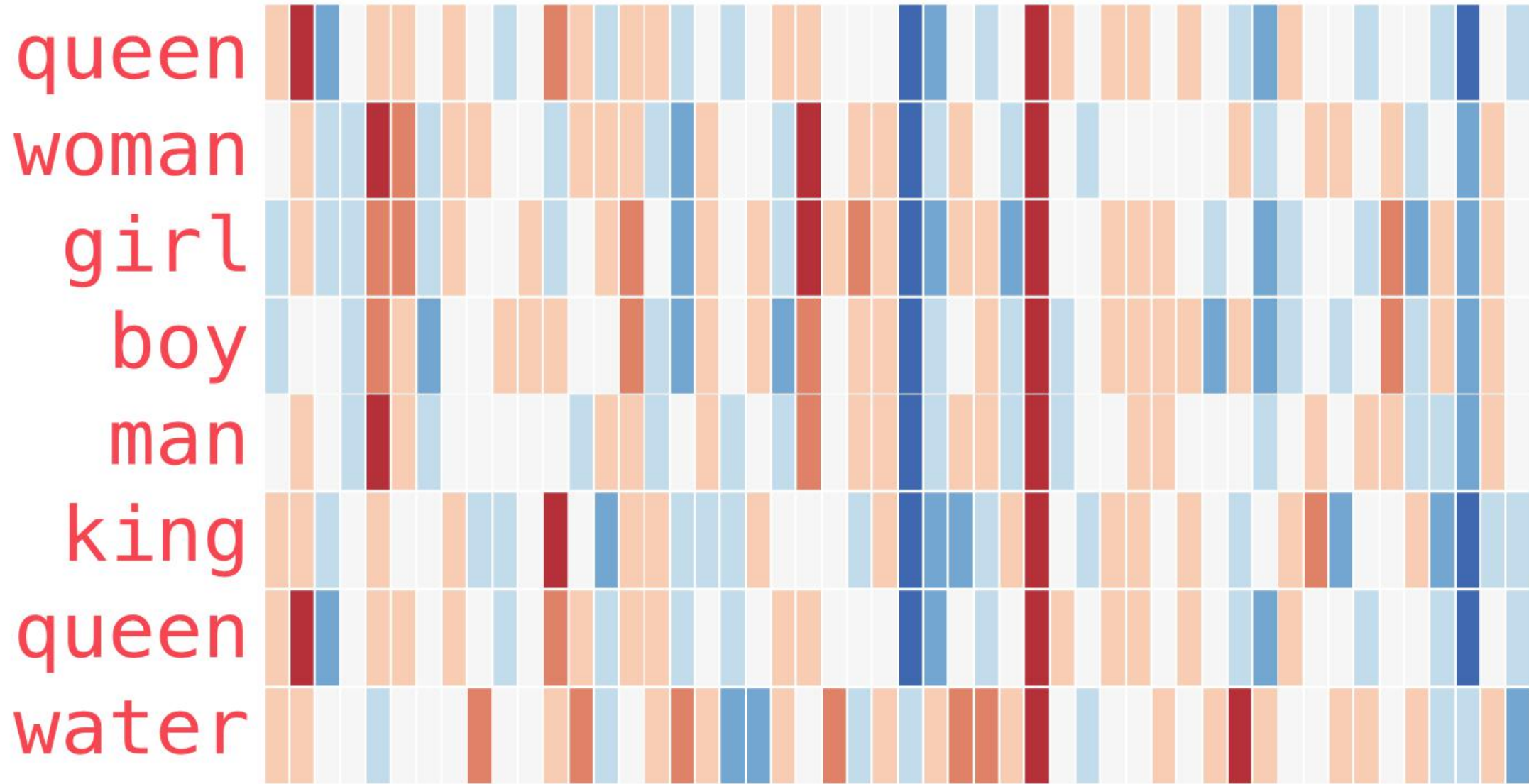


- “boy” and “girl” also have places where they are similar to each other, but different from “woman” or “man”
- Could these be coding for a vague conception of youth? possible.





- “king” and “queen” are similar to each other and distinct from all the others
- Could these be coding for a vague concept of royalty?



- Water is added to show the differences between categories – People and Object
- Blue column going all the way down and stopping before the embedding for “water”.

# Analogies

- Famous examples that show an incredible property of embeddings is the concept of analogies
- add and subtract word embeddings and arrive at interesting results
- 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)]
```

# Visualize this Analogy

king - man + woman  $\approx$  queen



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

# Training Process of Embeddings

- Conceptual parent of word embeddings: the neural language model
- to give an example of an NLP application, one of the best examples would be the next-word prediction feature of a smartphone keyboard
- It's a feature that billions of people use hundreds of times every day



# Language Modeling

- Next-word prediction is a task that can be addressed by a language model
- A language model can take a list of words (let's say two words), and attempt to predict the word that follows them.

# Language Modeling

- In the screenshot above, we can think of the model as one that took in these two green words (thou shalt)
- Returned a list of suggestions (“not” being the one with the highest probability):

input/feature #1

input/feature #2

output/label

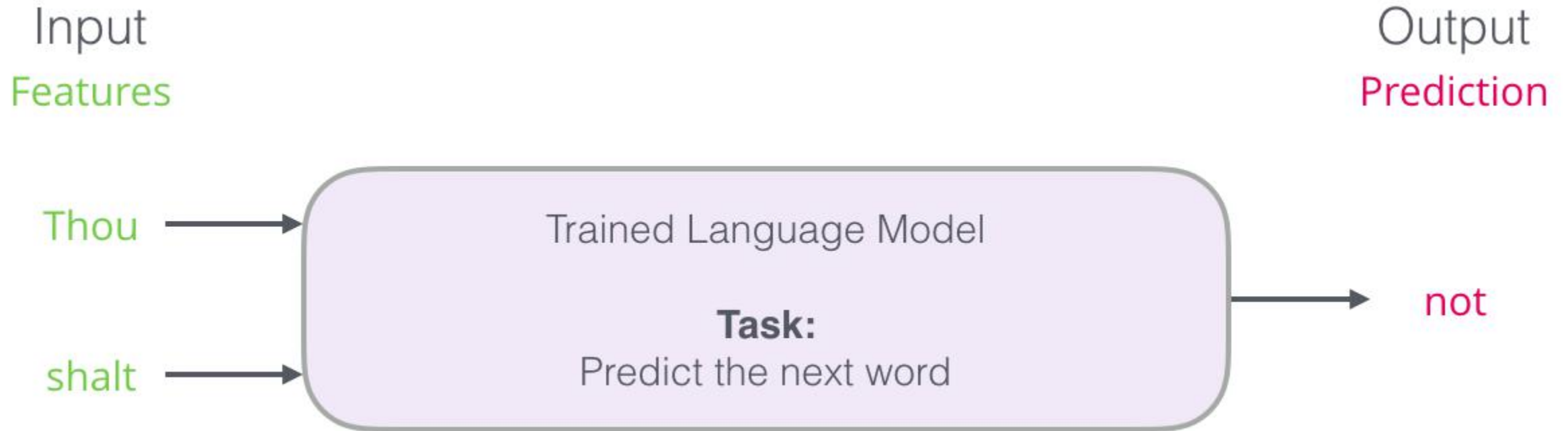
Thou

shalt

---

# Language Modeling

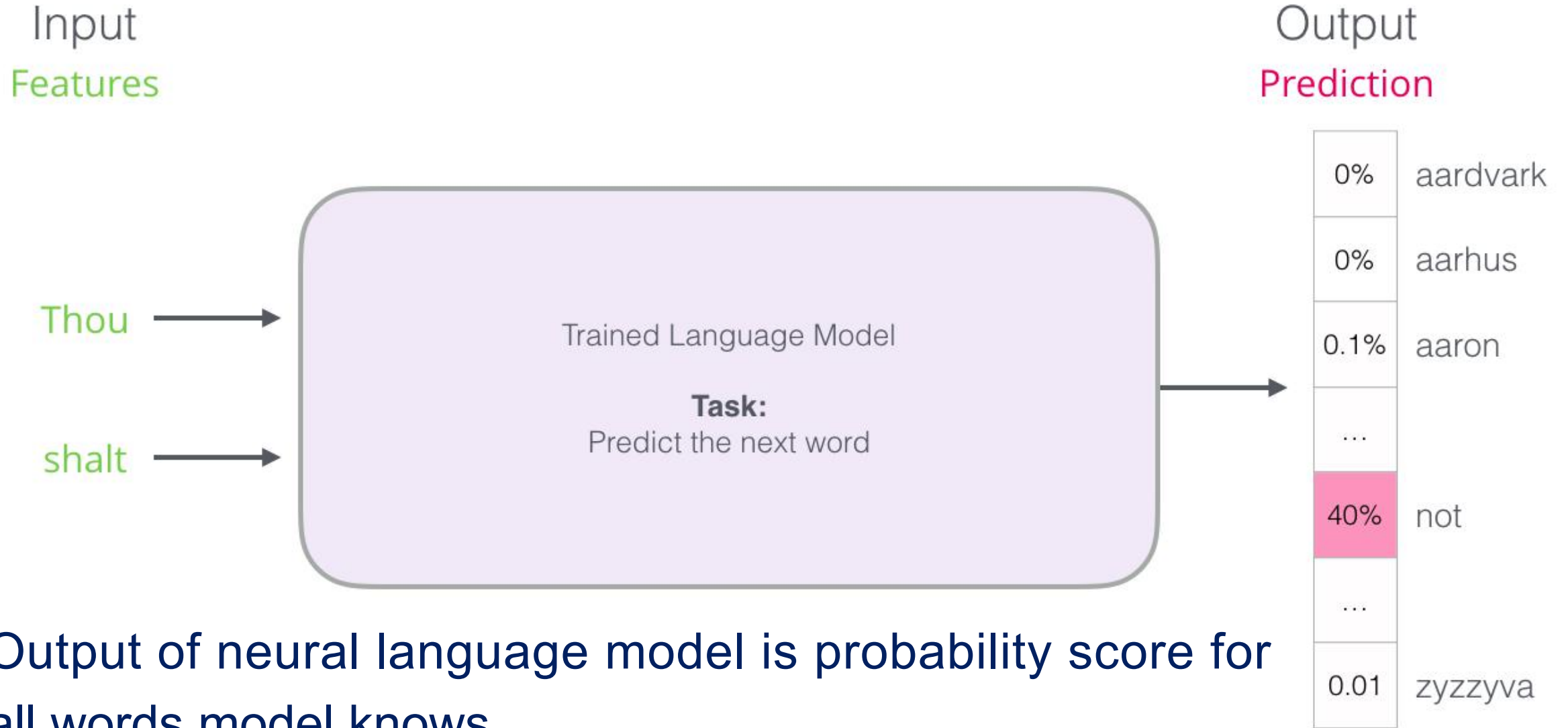
- We can think of the model as looking like this black box:





# Language Modeling

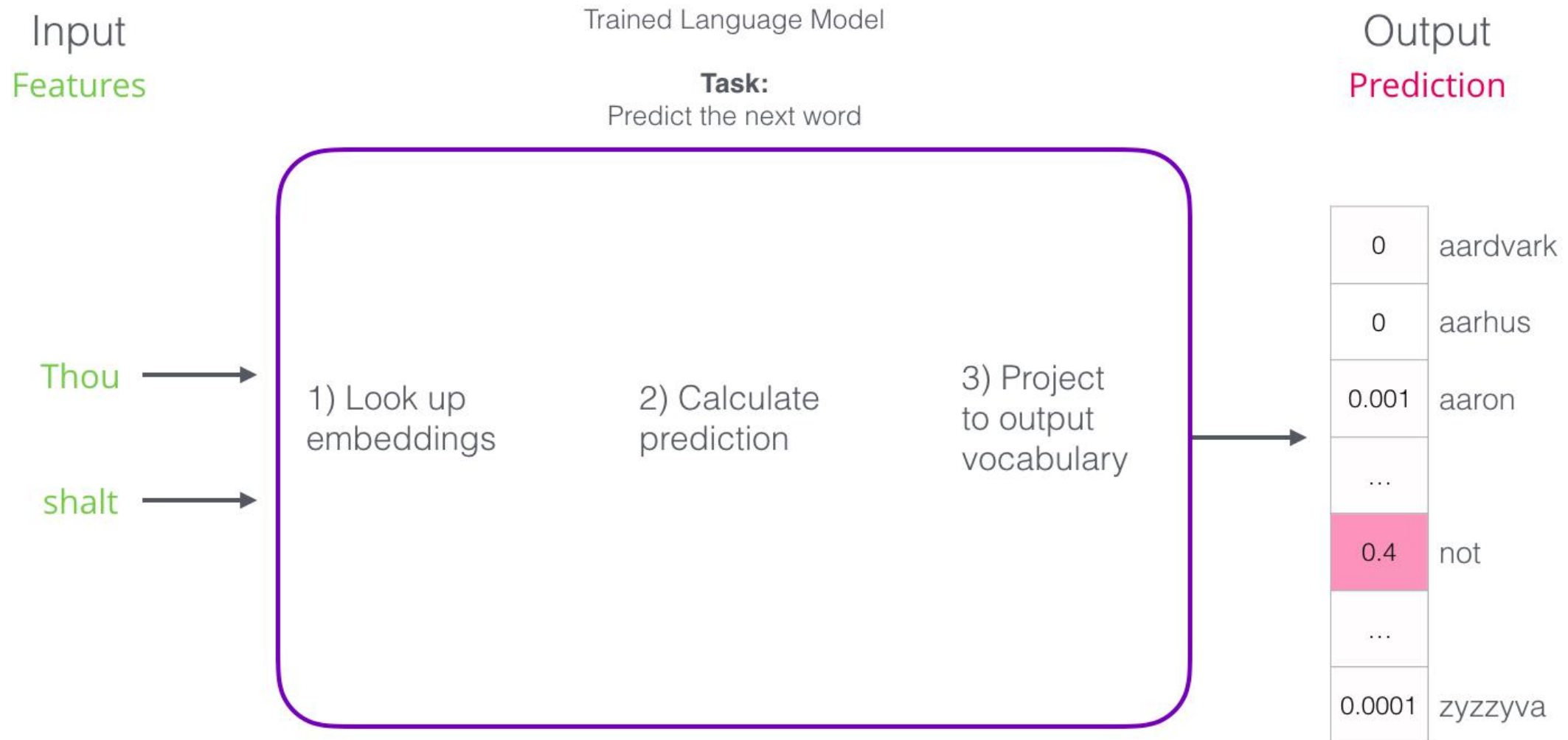
- But in practice, the model doesn't output only one word
- It actually outputs a probability score for all the words it knows
- the model's "vocabulary", which can range from a few thousand to over a million words
- The keyboard application then has to find the words with the highest scores, and present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



- Output of neural language model is probability score for all words model knows
- We're referring probability as a percentage
- 40% is 0.4 in the output vector

# Neural Language Modeling

- After being trained, early neural language models (Bengio 2003) would calculate a prediction in three steps:



Input  
Features

Trained Language Model

Output  
Prediction

**Task:**  
Predict the next word

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

# Language Model Training

- Language models have a huge advantage over most other machine learning models
- Advantage is that we are able to train them on running text – which we have an abundance of
- Think of all the books, articles, Wikipedia content, and other forms of text data we have lying around
- Contrast this with a lot of other machine learning models which need hand-crafted features and specially-collected data

# Language Model Training

- “You shall know a word by the company it keeps” J.R. Firth
- Words get their embeddings by us looking at which other words they tend to appear next to
- The mechanics of that is that
  1. We get a lot of text data (say, all Wikipedia articles, for example). then
  2. We have a window (say, of three words) that we slide against all of that text.
  3. The sliding window generates training samples for our model

# Language Model Training

- As this window slides against the text, we (virtually) generate a dataset that we use to train a model
- To look exactly at how that's done, let's see how the sliding window processes this phrase:
- “Thou shalt not make a machine in the likeness of a human mind” ~Dune

# Language Model Training

- When we start, the window is on the first three words of the sentence:

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	...
------	-------	-----	------	---	---------	----	-----	-----

Dataset

input 1	input 2	output



# Language Model Training

- We take the first two words to be features, and the third word to be a label:

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	...
------	-------	-----	------	---	---------	----	-----	-----

Dataset

input 1	input 2	output
thou	shalt	not

# Language Model Training

- We then slide our window to the next position and create a second

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	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make

# Language Model Training

- We have a larger dataset of which words tend to appear after different pairs of words:

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

# Language Model Training

- Models tend to be trained while we're sliding the window
- But I find it clearer to logically separate the “dataset generation” phase from the training phase
- Aside from neural-network-based approaches to language modeling, a technique called N-grams was commonly used to train language models
- Android keyboard, introducing their neural language model and comparing it with their previous N-gram model

# Look both ways

- Knowing what you know from earlier, fill in the blank:

Jay was hit by a \_\_\_\_\_

- Gave you here is five words before the blank word (and an earlier mention of “bus”)
- Guess
- Bus

# Look both ways

- One more piece of information
- a word after the blank
- Would that change your answer?

Jay was hit by a \_\_\_\_\_ bus

- Completely changes what should go in the blank
- Word red is now the most likely to go into the blank
- What we learn from this is the words both before and after a specific word carry informational value

# Look both ways

- Accounting for both directions (words to the left and to the right of the word we're guessing) leads to better word embeddings
- Let's see how we can adjust the way we're training the model to account for this.

# Continuous Bag of Words

- 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

- This architecture tries to guess neighboring words using the current word
- We can think of the window it slides against the training text as looking like this:

Jay was hit by a red bus in...



- The word in the green slot would be the input word, each pink box would be a possible output.

# Skipgram

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

Jay was hit by a red bus in...

by	a	red	bus	in
----	---	-----	-----	----

input	output
red	by
red	a
red	bus
red	in

# Visualize Skipgram

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

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word

# Skipgram

- This would add these four samples to our training dataset:

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

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

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

# Skipgram

- Slide our window to the next position:

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	...
------	-------	-----	------	---	---------	----	-----	-----

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

# Skipgram

- Which generates our next four examples:

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	...
------	-------	-----	------	---	---------	----	-----	-----

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

# Skipgram

- A couple of positions later, we have a lot more examples:

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

# Skipgram

- Now we have our skipgram training dataset that we extracted from existing running text
- Let's glance at how we use it to train a basic neural language model that predicts the neighboring word.



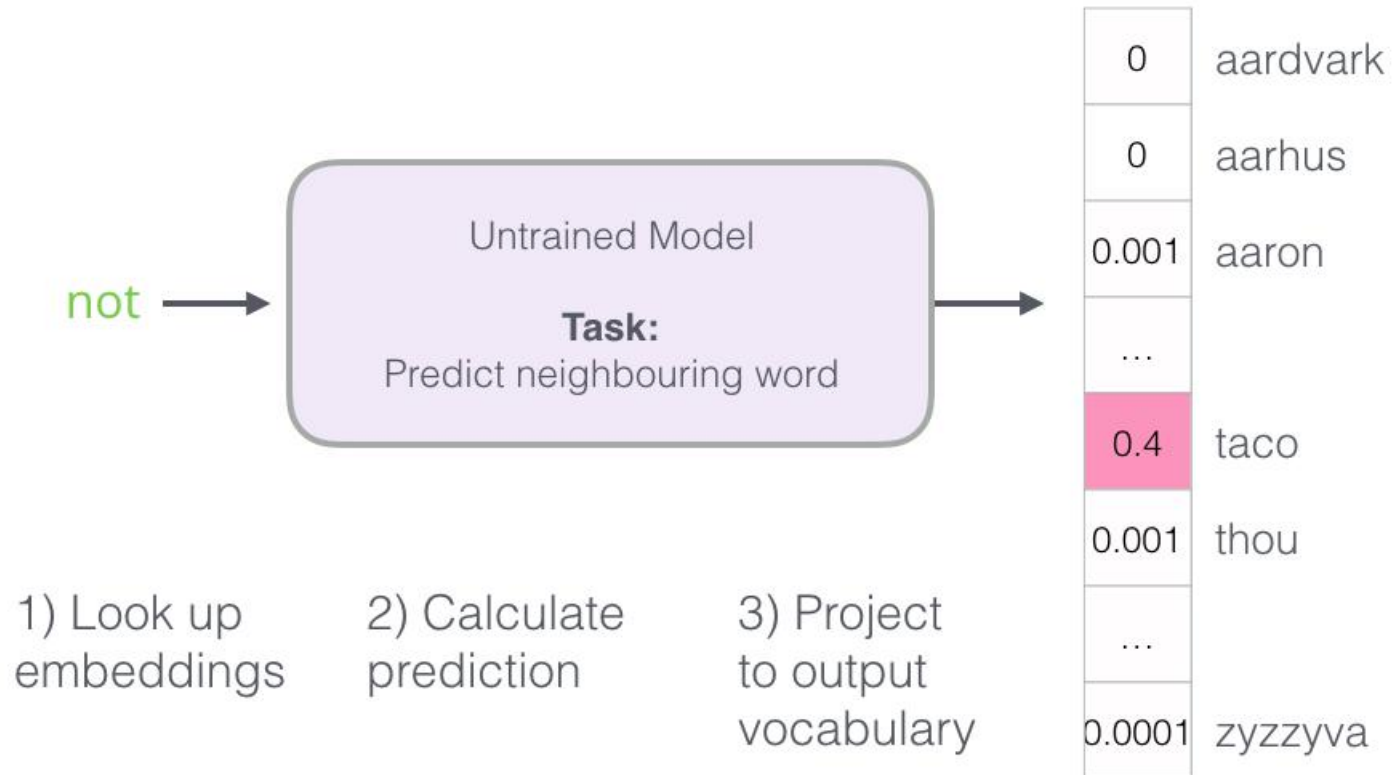
input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

not →



# Skipgram

- 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.



# Skipgram

- 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:

# Look both ways

- Accounting for both directions (words to the left and to the right of the word we're guessing) leads to better word embeddings
- Let's see how we can adjust the way we're training the model to account for this.

## Actual Target

0
0
0
...
0
1
...
0

-

## Model Prediction

0	aardvark
0	aarhus
0.001	aaron
...	
0.4	taco
0.001	thou
...	
0.0001	zyzzyva

- The 'target vector' is one where the target word has the probability 1, and all other words have the probability 0.

Actual  
Target

0
0
0
...
0
1
...
0

-

Model  
Prediction

0	aardvark
0	aarhus
0.001	aaron
...	
0.4	taco
0.001	thou
...	
0.0001	zyzzyva

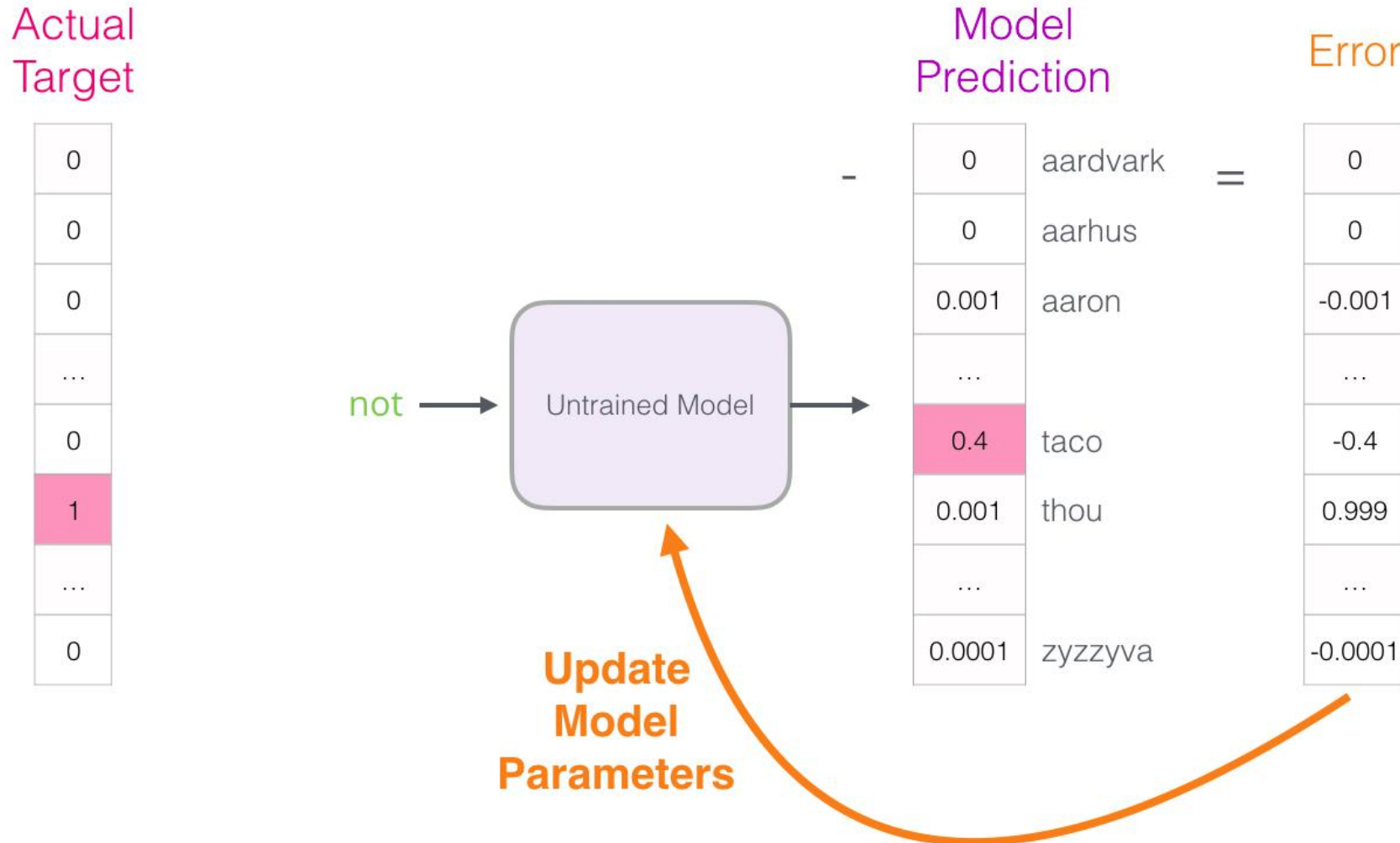
=

Error

0
0
-0.001
...
-0.4
0.999
...
-0.0001

- We subtract the two vectors resulting in an error vector
- 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.
- And that concludes the first step of the training.



# Training

- We proceed to do the same process with the next sample in our dataset and then the next
- Until we've covered all the samples in the dataset
- That concludes one epoch of training
- We do it over again for a number of epochs, and then we'd have our trained model and we can extract the embedding matrix from it and use it for any other application.



# Negative Sampling

- Recall the three steps of how this neural language model calculates its prediction:



1) Look up  
embeddings

2) Calculate  
prediction

**3) Project  
to output  
vocabulary**

**[Computationally  
Intensive]**

# Third step is very expensive

- From a computational point of view
- Especially knowing that we will do it once for every training sample in our dataset (easily tens of millions of times)
- We need to do something to improve performance.

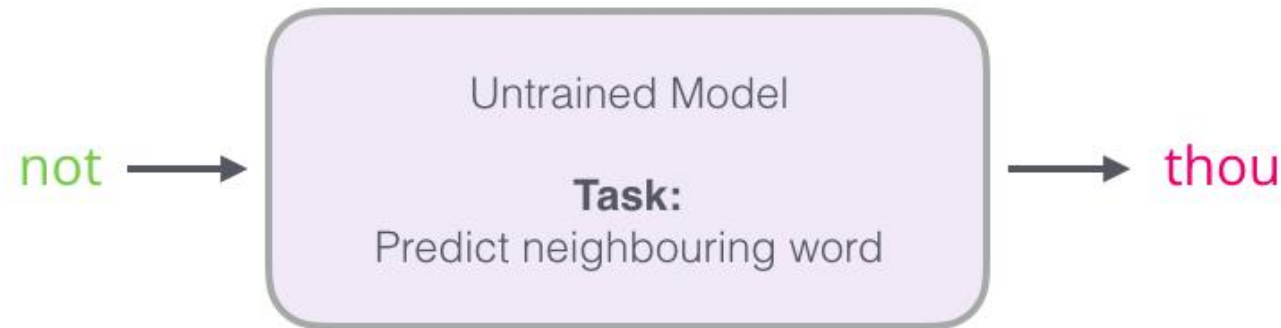
# Third step is very expensive

- One way is to split our target into two steps:
- Generate high-quality word embeddings (Don't worry about next-word prediction).
- Use these high-quality embeddings to train a language model (to do next-word prediction).

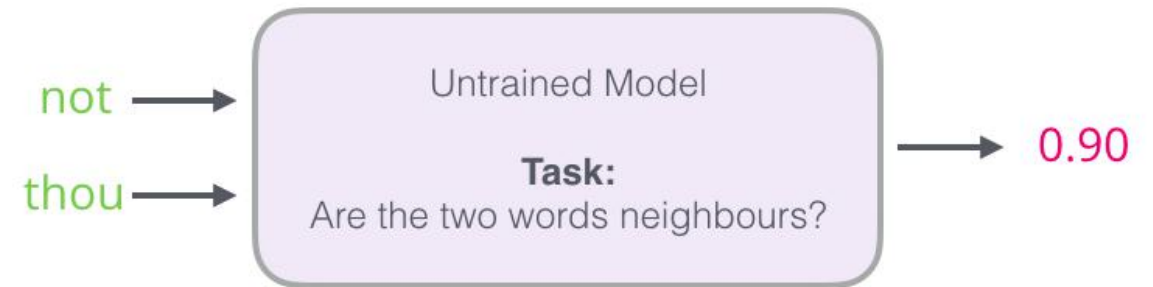
# Generate high-quality embeddings

- To generate high-quality embeddings using a high-performance model, we can switch the model's task from predicting a neighboring word
- to a model that takes input and output word, and outputs a score indicating if they're neighbors or not (0 - "not neighbors", 1 - "neighbors")

Change Task from



To:



## Third step is very expensive

- 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 our dataset – the label in a new column with values 0 or 1
- They will be all 1 since all the words we added are neighbors.

# Third step is very expensive

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	<b>1</b>
not	shalt	<b>1</b>
not	make	<b>1</b>
not	a	<b>1</b>
make	shalt	<b>1</b>
make	not	<b>1</b>
make	a	<b>1</b>
make	machine	<b>1</b>

## Third step is very expensive

- 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.

# Negative Examples

- To address this, we need to introduce negative samples to our dataset – samples of words that are not neighbors.
- Our model needs to return 0 for those samples.
- Now that's a challenge that the model has to work hard to solve – but still at blazing fast speed.



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)

## Skipgram

shalt	not	make	a	machine
-------	-----	------	---	---------

input	output
make	shalt
make	not
make	a
make	machine

## Negative Sampling

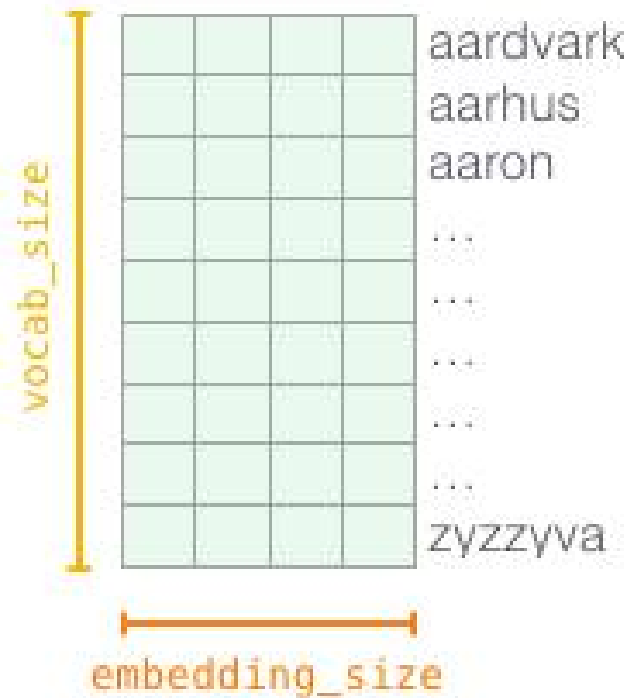
input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0

# Word2vec Training Process

1. Pre-process the text we're training the model against
2. Determine size of our vocabulary (we'll call this `vocab_size`, think of it as, say, 10,000) and which words belong to it.
3. Create two matrices – an Embedding matrix and a Context matrix.
  1. These two matrices have an embedding for each word in our vocabulary (So `vocab_size` is one of their dimensions).
  2. Second dimension is how long we want each embedding to be (`embedding_size` – 300 is a common value, but we've looked at an example of 50 earlier).

# Embedding and Context Matrices

## Embedding



## Context



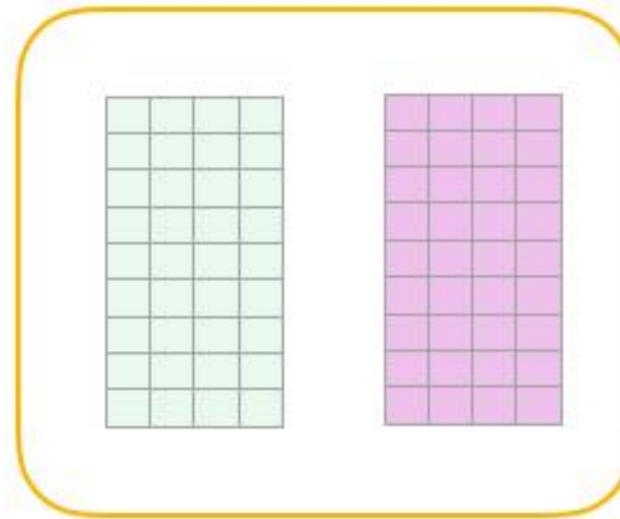
# Word2vec Training Process

4. Initialize these matrices with random values
5. In each training step, 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



# Word2vec Training Process

- 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 input word, look in Embedding matrix.
- For the context words, look in Context matrix (even though both matrices have an embedding for every word in our vocabulary).

# Embedding and Context Matrices

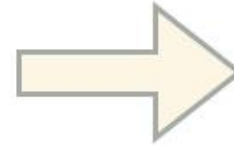
Embedding

				aardvark
				aarhus
				aaron
				...
				not
				...
				...
				...
				zyzzyva

Context

				aardvark
				aarhus
				aaron
				...
				taco
				...
				thou
				...
				zyzzyva

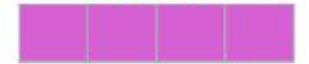
Look up  
embeddings



not



aaron



taco

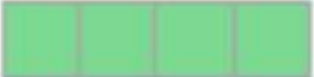







thou



# Word2vec Training Process

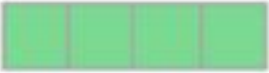





- Then, we take the dot product of input embedding with each of the context embeddings
- In each case, that would result in a number, that indicates the similarity of the input and context embeddings

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



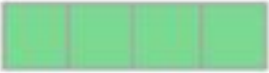





# Word2vec Training Process

- 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

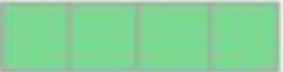


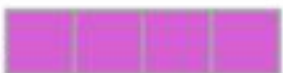


# Word2vec Training Process

- 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.

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






# Word2vec Training Process

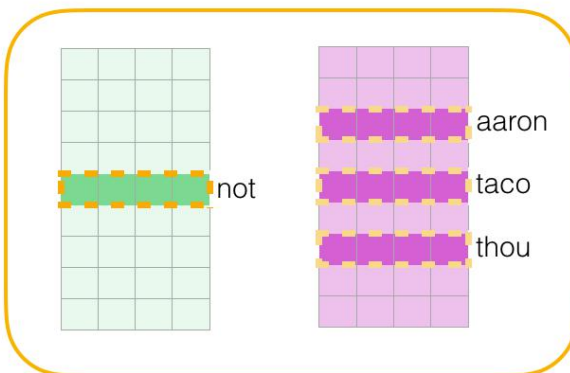
- Untrained model has made a prediction, and seeing as though we have an actual target label to compare against, let's calculate how much error is in the model's prediction
- Just subtract sigmoid scores from the target labels.

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

# Word2vec Training Process

- Here comes the “learning” part of “machine learning”.
- 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.

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



Update  
Model  
Parameters

# Word2vec Training Process

- This concludes the training step.
- We emerge from it with slightly better embeddings for the words involved in this step (not, thou, aaron, and taco).
- We now proceed to our next step (the next positive sample and its associated negative samples) and do the same process again.
- The embeddings continue to be improved while we cycle through our entire dataset for a number of times.
- We can then stop the training process, discard Context matrix, and use Embeddings matrix as our pre-trained embeddings for the next task.

# HyperParameters of Word2vec

- Two key hyperparameters in the word2vec training process are:
  - Window size
  - Number of negative samples
- Different tasks are served better by different window sizes
- One heuristic is that smaller window sizes (2-15) lead to embeddings where high similarity scores between two embeddings indicates that the words are interchangeable
- notice that antonyms are often interchangeable if we're only looking at their surrounding words – e.g. good and bad often appear in similar contexts)

# HyperParameters of Word2vec

- Larger window sizes (15-50, or even more) lead to embeddings where similarity is more indicative of relatedness of the words
- In practice, you'll often have to provide annotations that guide the embedding process leading to a useful similarity sense for your task.
- The Gensim default window size is 5 (two words before and two words after the input word, in addition to the input word itself).

# HyperParameters of Word2vec

Negative samples: 2

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

Negative samples: 5

input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0
make	finglonger	0
make	plumbus	0
make	mango	0



# HyperParameters of Word2vec

- The number of negative samples is another factor of the training process.
- The original paper prescribes 5-20 as being a good number of negative samples.
- It also states that 2-5 seems to be enough when you have a large enough dataset.
- The Gensim default is 5 negative samples.