

# Learning about AI/ML for Text, with Wordle!

**Nick Burch**

Berlin Buzzwords 2022

## Nick Burch

Director of Engineering



@Gagravarr

## Code, Sample Data, Slides

[github.com/Gagravarr/BBuzz22-ML\\_Text](https://github.com/Gagravarr/BBuzz22-ML_Text)

All code in slides is taken from here!



## Our talk today

- Not your typical AI talk...
- AI, Text and Wordle!
- AI and Text - how?
- Do we need an AI?
- "Simple" AI for text
- Neural Networks
- Making it better
- Further resources

**What is AI?**

**And ML?**

**And why now?**

## **AI - Artificial Intelligence**

## **ML - Machine Learning**

(Larry) Tesler's Theorem - "AI is whatever hasn't been done yet."  
Which makes... ML what we can do today!

First big "AI Bubble" was mid-1980s, over \$1 billion in AI startups that were generally touted as "expert systems"  
Two problems - not enough training data available, computers much more expensive than the experts they were trying to replace

## More expensive than the old experts?

Moore's Law to the rescue! One million dollars worth of 1985 computing power costs under one dollar today.

Amazon will rent you a machine with 1 TB of memory for roughly the cost of a latte per hour.

A 4 TB memory machine is about \$25 / hour, less if reserved!

A 6 TB memory machine is about \$55 / hour

A 48 TB memory machine is available, pricing not disclosed, but it exists...





<https://xkcd.com/1838/>

## No data or software?

We have a lot more data available to train our ML models on.

You may not have Google's 3.5 billion searches per day, or Facebook's 65 billion messages today, but your company is certainly generating more than a few punchcards of data...

Open Source libraries and frameworks for AI / ML make it easy to get started, mean you can focus on your problem, not the system.

<https://www.ft.com/content/c93725c4-5e34-4b3b-aae8-dd4c241abebd>

## There is no such thing as ‘data’

### *Benedict Evans*

Technology is full of narratives, but one of the loudest and most persistent concerns artificial intelligence and something called “data”.

AI is the future, we are told, and it’s all about data — and data is the future, and we should own it and maybe be paid for it. And countries need data strategies and data sovereignty, too. Data is the new oil.

This is mostly nonsense. There is no such thing as “data”, it isn’t worth anything, and it doesn’t belong to you anyway.

**<https://www.ft.com/content/c93725c4-5e34-4b3b-aae8-dd4c241abebd>**

Most obviously, data is not one thing, but innumerable different collections of information, each of them specific to a particular application, that can't be used for anything else.

For instance, Siemens has wind turbine telemetry and Transport for London has ticket swipes, and those aren't interchangeable. You can't use the turbine telemetry to plan a new bus route, and if you gave both sets of data to Google or Tencent, that wouldn't help them build a better image recognition system.

**You need to build your own models for your data**

## Lots of Tutorials

But most beginners ones focused on Numbers and/or Images

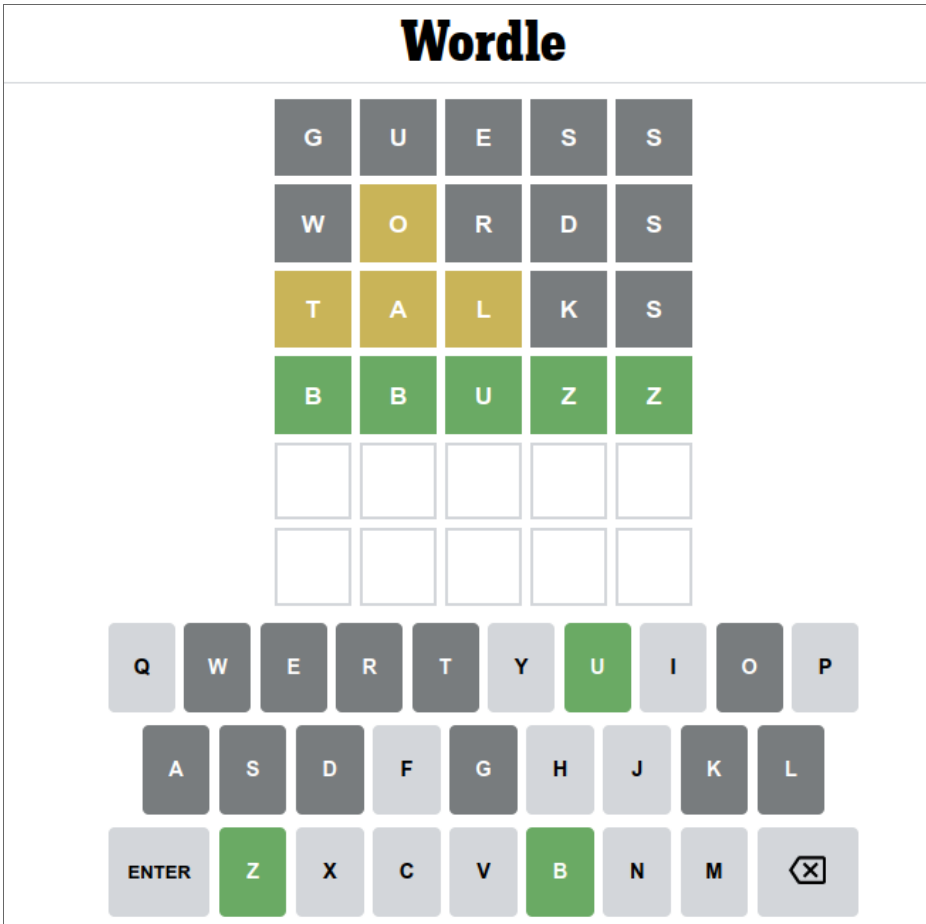
And most of the Text-related tutorials assume a fair amount of existing knowledge on ML and/or text analysis

Let's try to use this session to cover some basics, so you stand a chance of following the other tutorials and courses out there!

## **As we all have different data....**

There's no generic talk I can give to help everyone!

So let's pick a simple-ish area as our example, that you hopefully all know....



[nytimes.com/wordle](https://www.nytimes.com/wordle)



## In case you don't know Wordle

Simple, web-based word game, developed by Josh Wardle, and sold to the New York Times in 2022

6 attempts to guess a 5-letter word, with colours indicating if the letter is unused / wrong place / correct place

Now available (unofficially) in lots of different languages, and with many different variants!

Spoiler-free sharing via coloured squares, helped drive viral success

## You can get the word lists out of the JavaScript

But system dictionaries work pretty well too, and aren't cheating!

If you're working with other languages, remove accents

For a fun game, take care with plurals, past tense suffixes etc, but our AI won't mind those too much...

## Unix Tools to the Rescue!

```
#!/bin/bash
# Builds up wordle-like word lists from the system dictionaries
# To make it more worde-like, it:
# * Lowercases everything
# * Removes accents
# * Skips words with punctuation
# Unlike wordle, does include plurals, past tenses etc

DICTDIR="/usr/share/dict/"
DICTS="british-english french spanish"

for LANGUAGE in $DICTS; do
  DICT=${DICTDIR}${LANGUAGE}
  cat $DICT | \
    iconv -f utf8 -t ascii//TRANSLIT - | \
    tr '[:upper:]' '[:lower:]' | \
    grep '^[a-z][a-z][a-z][a-z][a-z]$\ ' > wordle/$LANGUAGE
done
```

```
for LANGUAGE in $DICTS; do
  echo $LANGUAGE
  echo `head -10 wordle/$LANGUAGE | tail -10`
  echo `sed -n '2000,2010p' wordle/$LANGUAGE`
  echo `sed -n '3000,3010p' wordle/$LANGUAGE`
  echo `tail -10 wordle/$LANGUAGE`
  echo ""
done

british-english
aaron abbas abdul abner abram abuja accra acruX acton acuff
brags braid brain brake brand brash brass brats brave bravo brawl
fours fowls foxed foxes foyer frack frail frame franc frank frats
zings zippy zombi zonal zoned zones zooms eclat epees etude

french
abaca abats abbes abces abeti abima abime abime ables aboie
emeut emiai emias emiat emiee emier emies emies emiez emirs emise
hales hales hales halez halez halle halls halos halte halva hamac
zones zones zonez zooma zome zome zooms zouka zouke zouke

spanish
abaca abaco abada abadi abajo abano abasi abate abati abece
goles golfa golfo golpe gomar gomel gomer gomia gonce gongo gorda
natri nauta naval navio nebel nebli nebri necia necio nefas negar
zumbo zumel zupia zurba zurda zurdo zureo zurra zuzar zuzon
```

## Do we need an AI at all?

**(This is a question you should be asking yourself in general for your problem!)**

## For Wordle, no...

Regular Expressions plus a dictionary of words does pretty well

Bayes Theorem plus frequency analysis plus normal stats techniques works very well

Shannon Entropy with Information Theory is pretty close

We're not doing Wordle + AI to make the best solver! We want to learn about AI for Text with Wordle

## Printing the squares in a few lines

```
result_green = "\U0001F7E9"
result_yellow = "\U0001F7E8"
result_white = "\u2B1C"
result_black = "\u25A0"

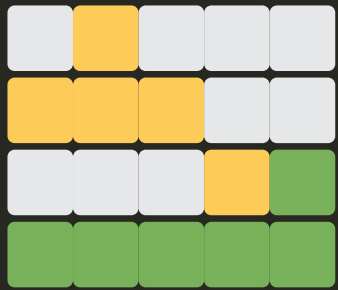
def calculate_squares(actual, guess):
    res = []
    for i in range(0,5):
        if guess[i] == actual[i]:
            res += result_green
        elif guess[i] in actual:
            res += result_yellow
        else:
            res += result_white
    return "".join(res)

def calc_with_squares(actual, guess):
    return calculate_squares(actual, guess) + " " + \
        " ".join( [ "%s\u20e3" % x for x in guess ] )
```



Let's give it a try!

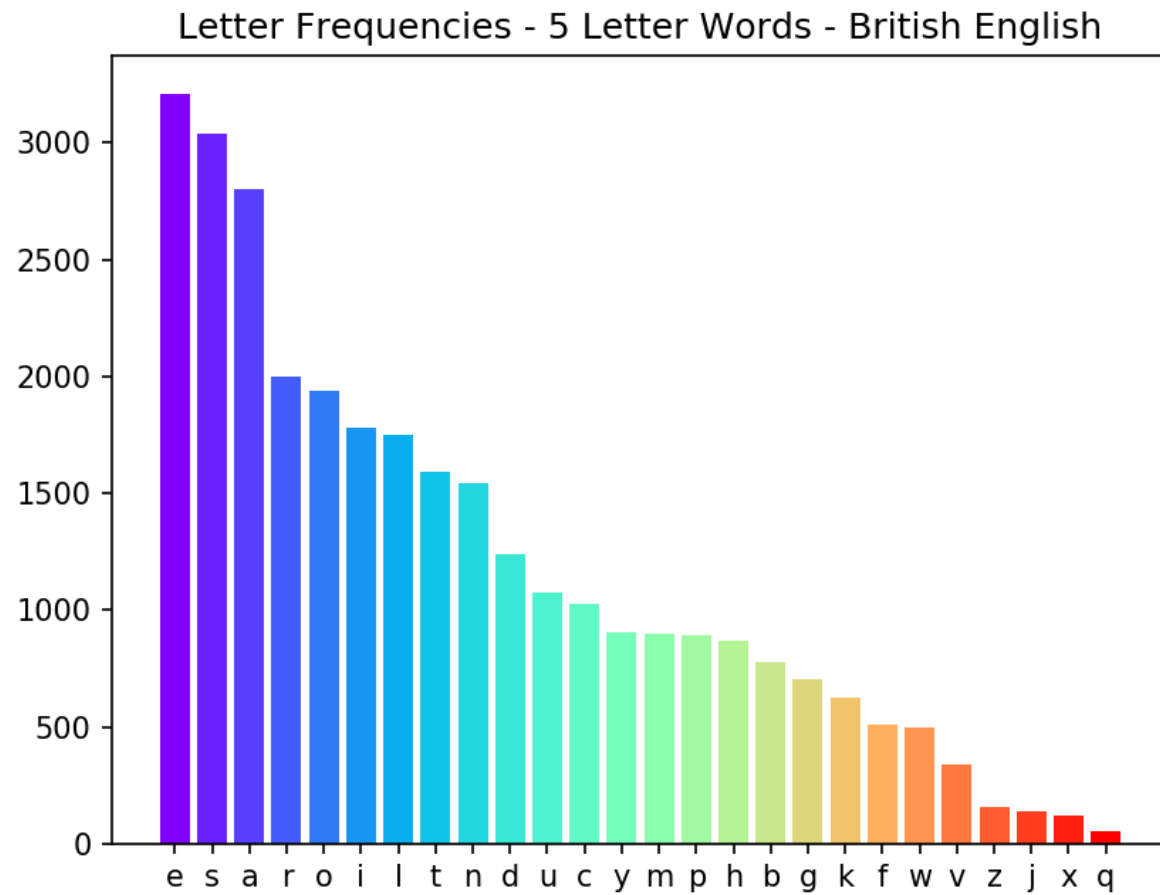
```
print(calc_with_squares("bbuzz", "guess"))  
print(calc_with_squares("bbuzz", "uzbek"))  
print(calc_with_squares("bbuzz", "soyuz"))  
print(calc_with_squares("bbuzz", "bbuzz"))
```

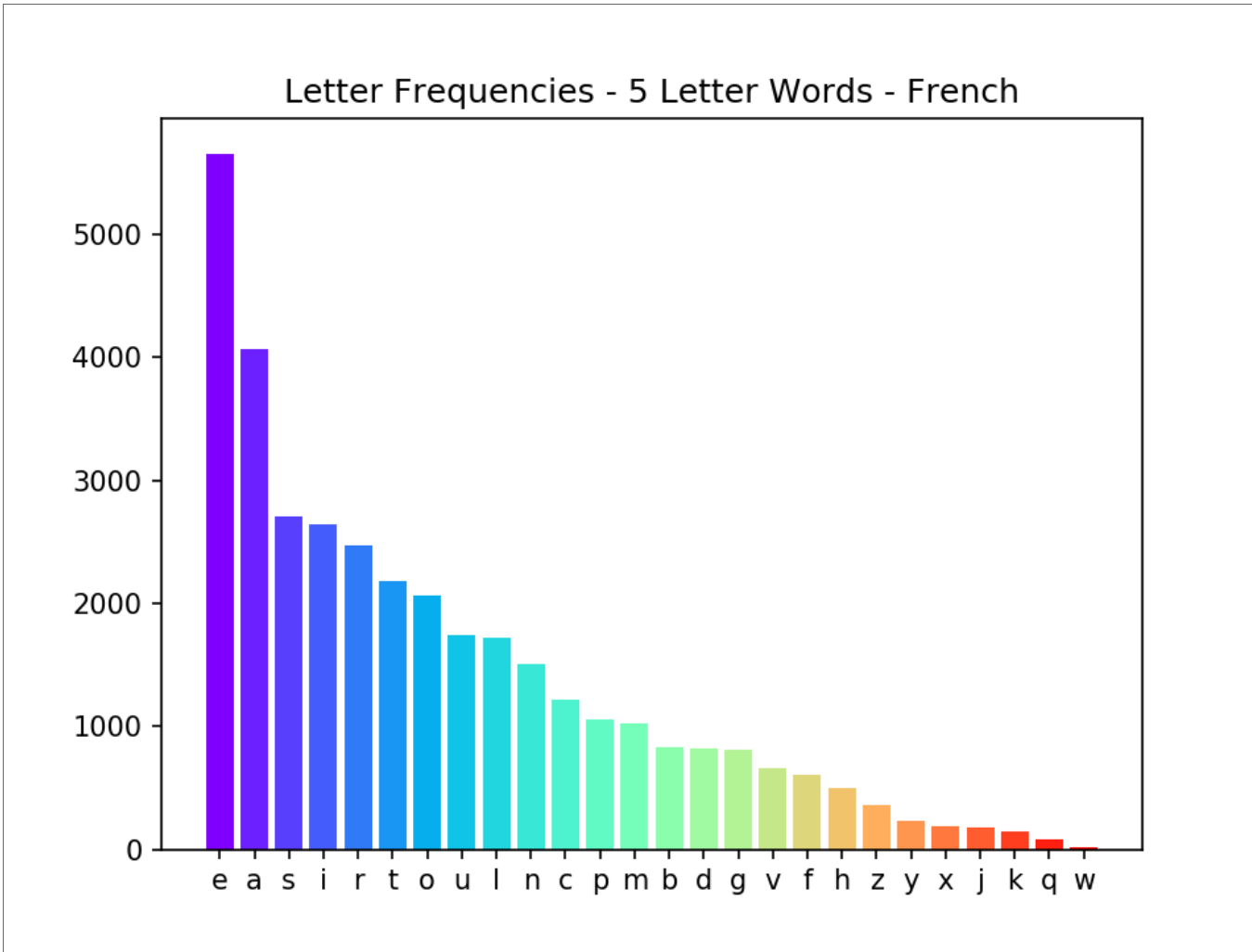


g	u	e	s	s
u	z	b	e	k
s	o	y	u	z
b	b	u	z	z

**Now we can score and print the boxes....**

**What does our word list look like?**





## How do we get those graphs?

```
words = pd.read_csv("wordle/british-english", header=0, names=["word"])
as_letters = words["word"].str.split('',n=5,expand=True).drop(0, axis=1)
print(as_letters.head(5))

   1  2  3  4  5
0  a  b  b  a  s
1  a  b  d  u  l
2  a  b  n  e  r
3  a  b  r  a  m
4  a  b  u  j  a

letter_counts = Counter(as_letters.values.flatten())
print(letter_counts.most_common(10))

[('e', 3210), ('s', 3037), ('a', 2803), ('r', 1996), ('o', 1938), ('i', 1783), ('l', 1751), ('t', 1589), ('n', 1545), ('d', 1236)]

ordered_letters = OrderedDict(letter_counts.most_common())
print(ordered_letters)

OrderedDict([('e', 3210), ('s', 3037), ('a', 2803), ('r', 1996), ('o', 1938), ('i', 1783), ('l', 1751), ('t', 1589), ('n', 1545), ('d', 1236), ('u', 1072), ('c', 1025), ('y', 904), ('m', 899), ('p', 891), ('h', 866), ('b', 774), ('g', 706), ('k', 627), ('f', 511), ('w', 496), ('v', 340), ('z', 158), ('j', 137), ('x', 119), ('q', 52)])

letter_colors = plt.cm.get_cmap("rainbow")(np.linspace(0,1,len(letter_counts)))
plt.bar(ordered_letters.keys(), ordered_letters.values(), color=letter_colors)
```

## A few key Python libraries to know about

Pandas

Numpy

Matplotlib

SciKitLearn

Python Collections

**We can load data, draw coloured boxes, are we good to go?**



## ML techniques don't like letters and words...

## ML needs a matrix of numeric values

Ideally mostly -1.0 to 1.0 or 0.0 to 1.0

One value for each *feature* of the thing to learn / predict on

Can be sparse (only a few non-zero values) or dense (mostly non-zero)

More features requires more memory and more CPU, so a tradeoff!

How can we turn our text into something like that?

## An aside - some terminology

**Feature** - An input variable, a value for one aspect of the thing to predict. eg height / weight / 1st RGB channel

**Label** - The value to be predicted / trained for, eg Dog / Cat / price of house the features describe

**Training** - Creating / learning a model to map from the features to the label

**Inference** - Applying the model to something new to get a prediction

<https://developers.google.com/machine-learning/crash-course/framing/ml-terminology>

## An aside - some terminology

**Regression** - A model to predict continuous values. eg "Given the location and number of bedrooms, what's the likely price of a house"

**Classification** - A model to predict discrete values. eg "Is this an image of a Dog, a Cat or a Wombat?"

**Clustering** - A model to group similar things together. If those are labelled, it's classification. If those aren't labelled, it's clustering, and relies on unsupervised machine learning

<https://developers.google.com/machine-learning/clustering/overview>

## If we were dealing with lots of text

If we had sentences, phrases, full-text search documents etc

*One-Hot encoding* - 1 if present, otherwise 0

*CBOW Count* - how many times across the *continuous bag of words* each term occurs. (This loses position information though)

*TF-IDF* - term frequency, inverse document frequency. Down-weight common words, weight rarer terms higher, adjust for document length

See my talk from last year's conference for more on all this!

**Now, back to Wordle...**

## What would make a good starting word?

We want words with as many popular letters as possible!  
Can we figure that out?

```
def score_by_letter_counts(wordrow):
    word = wordrow["word"]
    return np.product(
        [letter_counts[l]*1.0/max_letter_count for l in word] )

words_letterscore = words.copy(deep=True)
words_letterscore["score_all"] = words_letterscore.apply(
    lambda x: score_by_letter_counts(x,False), axis=1)
words_letterscore.head(5)
words_letterscore.sort_values("score_all",ascending=False).head(5)
```

	word	score_all
0	abbas	0.041942
1	abdul	0.014769
2	abner	0.063013
3	abram	0.032017
4	abuja	0.002620

	word	score_all
2700	eases	0.781623
1700	asses	0.739499
1154	reese	0.588295
4896	seers	0.556590
2761	erase	0.513704



## Is *Eases* a good starting word?

Double letters are actually bad to start, because we don't get as much information out  
First word isn't just about the greens, it's also about the yellows  
If we exclude words with double letters, what then?

## This time excluding words with duplicate letters

```
# Removes any duplicate letters, may also scramble order
remove_duplicate_letters = lambda word: "".join(set(word))

def score_by_letter_counts(wordrow, skip_duplicates=False):
    word = wordrow["word"]
    if len(remove_duplicate_letters(word)) != 5:
        return 0
    return np.product(
        [letter_counts[l]*1.0/max_letter_count for l in word] )

words_letterscore["score_nodup"] = words_letterscore.apply(
    lambda x: score_by_letter_counts(x), axis=1)
render(words_letterscore.sort_values("score_nodup", ascending=False).head(5))
```

	word	score_all	score_nodup
1684	arose	0.310143	0.310143
4599	raise	0.285338	0.285338
1681	arise	0.285338	0.285338
26	aires	0.285338	0.285338
83	aries	0.285338	0.285338

## **Are *Raise* and *Arise* the same?**

Full stats analysis would compute and compare frequencies taking account of the letter position  
Other talks are available on the best way to solve Wordle with stats!

## How about TF-IDF?

Lots of talks at Buzzwords about search scoring

TF-IDF, BM-25, more advanced techniques

Would they help our approach?

Handily, SciKitLearn has TF-IDF support available

```
tfidf = TfidfVectorizer(sublinear_tf=True, min_df=0, stop_words=None,
                        analyzer="char", ngram_range=(1,1))
tfidf_matrix = tfidf.fit_transform(words["word"])

print(tfidf.get_feature_names())

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
 'w', 'x', 'y', 'z']

words_letterscore["score_tfidf"] = pd.Series([
    tfidf_matrix[idx].sum()/5 for idx in range(len(words)) ])
words_letterscore.sort_values("score_tfidf",ascending=False).head(5)
```

	word	score_all	score_nodup	score_tfidf
2219	chump	0.002236	0.002236	0.447000
2661	duchy	0.003120	0.003120	0.446862
2058	bumpy	0.001763	0.001763	0.446853
2673	dumpy	0.002815	0.002815	0.446760
4039	mucky	0.001643	0.001643	0.446317

## Chump, Duchy, Bumpy, Dumpy, Mucky

IDF - inverse document frequency - has hurt us here

For a starting word, we want common letters, but IDF has boosted ones there aren't in that many words

Not terrible - no Zs - but not great

We want to narrow down the possible answer space as quickly as possible, and these won't usually help as much

## Understand your data

You need to understand your data

You need to understand what's good and what's bad

Just because you can get a score, doesn't mean it's the right one...

## Our first AI Bot - Multinomial Naive Bayes

Classifier which can predict from learning what's similar

Feed it all the words as a matrix, eg TF-IDF of letters

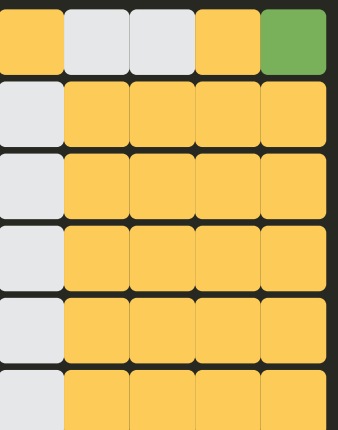
Can give it a word, and it will suggest similar ones

Can give it some letters, get "good" words based on those



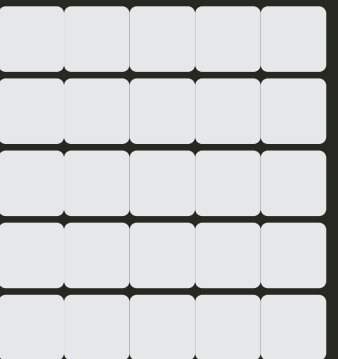
The image shows a 10x5 grid of colored squares on the left, with a list of words on the right. The grid uses yellow for correct letters in the correct position, white for correct letters in the wrong position, and green for incorrect letters. The words listed are: arose, abbas, alana, abbas, alana, abbas, alana, abbas, alana, abbas.

Yellow	White	White	Yellow	White	arose
Yellow	White	White	Green	Yellow	abbas
Yellow	White	Yellow	Yellow	Yellow	alana
Yellow	White	White	Green	Yellow	abbas
Yellow	White	Yellow	Yellow	Yellow	alana
Yellow	White	White	Green	Yellow	abbas
Yellow	White	Yellow	Yellow	Yellow	alana
Yellow	White	White	Green	Yellow	abbas
Yellow	White	Yellow	Yellow	Yellow	alana
Yellow	White	White	Green	Yellow	abbas



abbas  
lassa  
lassa  
lassa  
lassa  
lassa

You didn't **get it** right! Word was grams



abbas  
abbas  
abbas  
abbas  
abbas

You didn't **get it** right! Word was icing

## It's pretty terrible...

Classifier gets stuck

Classifier - only ever a single answer

Doesn't understand about misses

Doesn't know about letter positions

Very similar words are bad in wordle!

Wordle				
R	A	I	S	E
T	H	O	S	E
C	L	O	S	E
G	O	O	S	E

The kinds of wordles I hate... similar words aren't good!

## K-Means Clustering

Would help us find similar words, clustering groups together

Approach would let us find the best number of clusters, we can use eg Silhouette Coefficient to test the best trade-off between different cluster sizes

However, we still don't want lots of similar words...

## Some other terms to be aware of

## Embeddings and Feature Extraction

Even from just a few hundred talks, our TF-IDF had a lot different terms in it. However, most talks only use a subset of those terms, so lots of TF-IDF matrix values are 0. Our TF-IDF is *sparse*

Bayse and K-Means, amongst others, are fine with sparse matrices. Other techniques, eg Neural Networks, need *dense* matrices, where most values are non-zero

Using stop-words and stemming helps a little bit here, but we need a few more orders of magnitude change!

[https://scikit-learn.org/stable/modules/feature\\_extraction.html#text-feature-extraction](https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction)

<https://developers.google.com/machine-learning/crash-course/embeddings/obtaining-embeddings>

## Embeddings and Feature Extraction

TF-IDF has no semantic information

Word order matters - information depends on the context

The place of the word in the sentence makes a big difference!

*My kindle is easy to use, I do not need help*

*I do need help, my kindle is not easy to use*

And with our terrible MNB bot, where in the word the letter goes!



## Testing, Cross Validation

When we know what our model should be predicting for a set of input data (eg what the human-provided labels are for a bunch of input features), we need to measure how well the model does!

Train on one set of known-data, then predict against another set, and see how close the predictions came to what we know are the answers

Our model might do *too well*, if it ends up learning some specific bits of our training data, this is *over-fitting*

Generally we want a train / test split, and maybe validate too, which should all be representative of the data we'll predict with

If we don't have much data available, we can train the model several times with different segments as train/test, and ensure always similar accuracy

## Hyper-parameters and tuning

A hyper-parameter is anything we tune / select / change in our ML, that's independent of the data and of the features selected

eg k-means, that includes the range of clusters to try fitting to

Often relate to seeds, steps, number of clusters / layers, how much to change between iterations, and how much to leave alone

Pick the wrong parameters, and your model might get stuck in a local minima a long way off the best answer, or might take ages to converge, or may never even complete!

Along with identifying appropriate features, and cleaning your data, selecting appropriate hyper-parameters is a tough bit of data science

## Errors

For a binary classification, there are 4 possible states: True Positive, True Negative, False Positive and False Negative

What to aim for depends on your problem, and the distribution of your data

eg if detecting cancer, is it better to give someone the all-clear when they actually have cancer (false negative)? Or to send them for treatment that they don't need (false positive)

**Precision** is ratio of correct values in our results, **recall** is ratio correct-found to correct-all. **Confidence** is how well the model thinks it did.

## Biases

If your input data is biased, the model will be biased

If you try to hide some biases from your model, it might still find them from other features.

eg hide Gender, but leave in Name. eg hide Race, but leave in postal code / zipcode, or first / elementary school

Be aware of your data biases, be aware of how people will use your model, try to re-weight your models to counteract biases

Be aware of implicit biases in your data, and impact if model is fed data from somewhere else / something else

## **BERT, Word2Vec, ELMo**

Word Embeddings, fairly small vector space (few hundred dimensions)

Similar words located nearby, direction through vector space carries meaning

Can do reasoning - Berlin is to Germany as Paris is to ???

Big part of semantic / neural search, see most of yesterday's talks!

```
from mxnet import nd
from mxnet.contrib.text import embedding

glove = embedding.GloVe(pretrained_file_name='glove.6B.50d.txt')

def find_nearest(vectors, wanted, num):
    # 1e-9 factor is to avoid zero/negative numbers
    cos = nd.dot(vectors, wanted.reshape((-1,))) / (
        (nd.sum(vectors * vectors, axis=1) + 1e-9).sqrt() *
        nd.sum(wanted * wanted).sqrt())
    top_n = nd.topk(cos, k=num, ret_typ='indices').asnumpy().astype('int32')
    return top_n, [cos[i].asscalar() for i in top_n]

def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed.get_vecs_by_tokens([token_a, token_b, token_c])
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = find_nearest(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[topk[0]] # Remove unknown words

def print_analogy(token_a, token_b, token_c, embed):
    anal = get_analogy(token_a, token_b, token_c, embed)
    print("The analogy for %s -> %s of %s is %s" %
          (token_a, token_b, token_c, anal))

print_analogy('berlin', 'germany', 'paris', glove)
```

## Can this help us solve Wordle?

No

We know what letters we do / don't need in our word

We don't know anything about what it means....

## Can this help with anything else?

Semantle - <https://semantle.com/>

We can re-implement this in a few lines of Python with Apache MXNet!

Also very helpful for improving relevancy in search, if done right...



## Are any ML techniques of use

Re-inforcement Learning

<https://pypi.org/project/gym-wordle/>

## Some other Wordle solving resources

<http://diamondgeezer.blogspot.com/2022/01/wordle-stats.html>

<https://medium.com/@tglaiel/the-mathematically-optimal-first-guess-in-wordle-cbcb03c19b0a>

<https://aperiodical.com/2022/02/a-mathematicians-guide-to-wordle/>

<https://www.mynl.com/static/blog/wordle.html>

## Code, Sample Data, Slides

[github.com/Gagravarr/BBuzz22-ML\\_Text](https://github.com/Gagravarr/BBuzz22-ML_Text)

All code from slides, plus additional bits

