UCLA Extension - Introduction to Data Science
COM SCI X 450.1
Author: Nathan Strong
Instructor: Ali El-Annan
Date: May 2020

Part 1: Comparing the databases Microsoft SQL Server (SQL) and Google Firestore (NoSQL)

Part 2: Comparing algorithms Naive Bayes (supervised) and K-Means Clustering (unsupervised)

Forward

This essay will compare and contrast two popular databases as well as two popular algorithms. The code and images in the following examples is all mine and can be downloaded here: https://github.com/NaughtyPhoton/ucla-data-science

Python code indentation may have shifted during PDF export. In some cases, the output beneath the code is more important than the code itself.

The Study -- What is the Most Popular Programming Language as Decided by Twitter

As a means of displaying the databases and algorithms in action, I will be doing a "study" of preferred programming languages as decided by Twitter. This study is merely a vehicle to depict the technology in question.
The programming languages I would like to examine are the 5 "Most Loved" programming languages of 2019 as voted by the Stack Overflow Community in their Annual Developer's Survey: https://insights.stackoverflow.com/survey/2019

In [73]:
```
from typing import Tuple

# In order of most loved -> least loved
SO_MOST_LOVED: Tuple[str] = ('Rust', 'Python', 'TypeScript',
'Kotlin', 'WebAssembly')
```

Collecting Tweets with Twython

Using the Python library Twython I can search for Tweets. The free tier of the Twitter Developer API only allows us to return 100 Tweets per request and can only go back 30 days. However, by using a feature of the API called Streaming, you can collect Tweets with a query as they are created. I've already extended the TwythonStreamer class in another file to compress this document. Below, I'll use my CustomStreamer class to collect Tweets with our language's names in them and filter them into a dictionary.

In [256]:
```
from typing import List, DefaultDict
from collections import defaultdict
import time
from custom_streamer import CustomStreamer
from tweet import Tweet

MINUTES_TO_LISTEN = 60 * 2

tweets_dict: [str, List[Tweet]] = defaultdict(list)
twitterStreamer = CustomStreamer(minutes_to_listen=MINUTES_TO_LISTEN)
twitterStreamer.statuses.filter(track=', '.join(SO_MOST_LOVED))

for _tweet in twitterStreamer.tweets:
    for language in SO_MOST_LOVED:
        if language.lower() in _tweet.text.lower():
            tweets_dict[language].append(_tweet)

for k, v in tweets_dict.items():
    print(f'{k}: {len(v)}')
```

Python: 949
Rust: 197
TypeScript: 64
Kotlin: 23
WebAssembly: 13

I can print the last couple of Python Tweets.

In [66]:
```
for x in tweets_dict['Python'][-2:]:
    print(f'~~~ {x.text}')
```

~~~ RT @machinelearnflx: 5 Best Courses to Learn Python’s Pandas Libary for Data Analysis and Data Scienc
e https://t.co/LeNNeTsCIc #DataScience
~~~ my knowledge on Python is poor

Part 1 - Databases

It's a good idea to store my collection of tweets in a database, as the Twitter API only allows for a limited amount of API calls. This way I can also add to my Tweet collection by running this script on different days, thus growing my sample size, and when I feel my collection is sufficient, I can stop querying for new Tweets.

Part 1.1 - SQL Databases

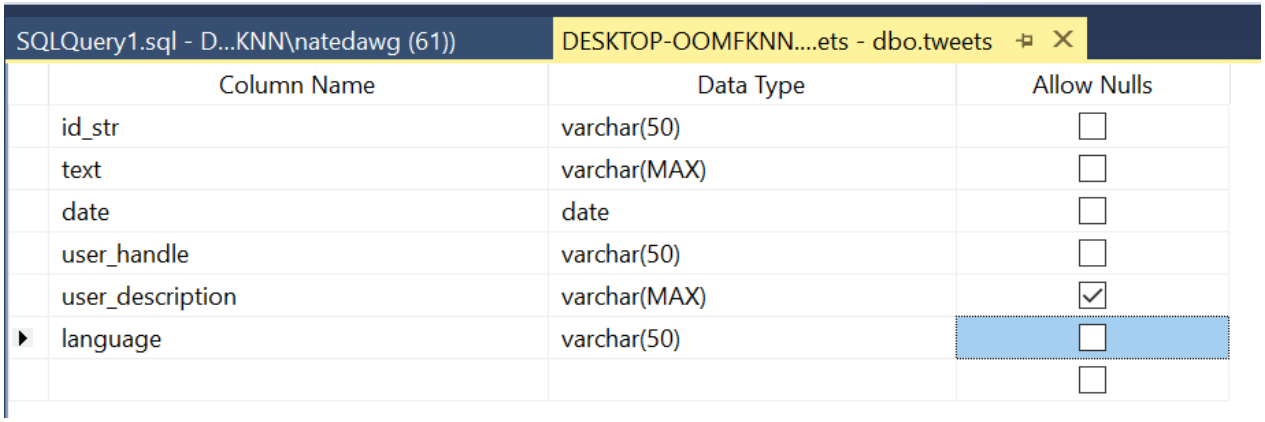
What is an SQL Database?

SQL stands for Structured Query Language; the language used to work with a relational database. A relational database is one which finds data based on its relationship to other data. The SQL language is (pretty much) the same between different DBMS (Database Management Systems).

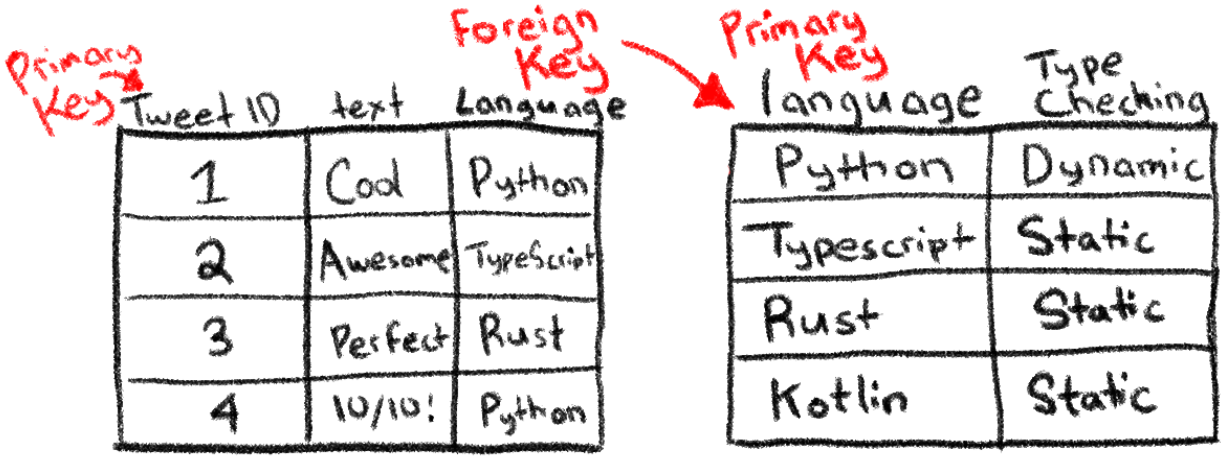
Microsoft SQL Server

I chose to use the Microsoft SQL Server for this project because I have never used it before and people seem to really like it. It uses it's own flavor of SQL called "transact-sql" (t-sql). It has an integrated envioronment called Microsoft SQL Manager which provides GUI tools for managing databases. The Microsoft SQL Server can also live on a Microsoft Azure instance if the database is designed to work on the cloud. It has a ton of other features, but notably (for data science) it has a service called SQL Server Analysis Services (SSAS) that provides tools for Machine Learning and Data Analysis (I won't be using these).

First, I'll define my table, columns, and data types in the GUI app, but this can also be done directly through SQL as well.



Since SQL is a relational database, we'll need a "foreign key", in my table which is a key that matches the "primary key" of another table so the data can be easily combined. In my case, I am only going to create this one table, but if I were to continue with this experiment, I would make the "language" column a foreign key so that I could make a seperate collection of languages and easily qeuery all the tweets for each language.



Using Python with Microsoft SQL Server - pyodbc

The easiest way to update my SQL server is with the Python library pyodbc. pyodc is an open-source Python library which provides CRUD (Create, Read, Update, Delete) methods for ODBC (Open Database Connectivity) enabled DBMS.

In [67]:
```
import pyodbc

# Connect to SQL Server
connect_string = 'Driver={SQL Server Native Client RDA 11.0}; \
                  Server=DESKTOP-OOMFKNN; \
                  Database=programming_tweets; \
                  Trusted_Connection=yes;'

with pyodbc.connect(connect_string) as conn:
    cursor = conn.cursor()

    # Define the columns and values to be updated
    insert_query = """INSERT INTO tweets (id_str, text, date, user_handle,
 user_description)
                      VALUES (?, ?, ?, ?, ?)"""

    # Loop through all the tweets
    for language, tweets in tweets_dict.items():
        for _tweet in tweets:
            # Define the new values to insert
            values = (_tweet.id_str, _tweet.text, _tweet.date, _tweet.user
_handle, _tweet.user_description)

            # Insert the new values into the database
            cursor.execute(insert_query, values)

    # Commit the inserts
    conn.commit()
```

Now that the database has been updated, we can read the information inside of it via a 'SELECT' query. Below, I am printing the last few tweets in the collection.

In [76]:
```
with pyodbc.connect(connect_string) as conn:
    cursor = conn.cursor()

    # Grab all the rows in the tweets table
    cursor.execute('SELECT user_handle, text FROM tweets')

    [print(f'user: {x[0]}\ntweet: {x[1]}\n') for x in list(cursor)[:2]]
```

user: Tall_Individual
tweet: @RunningEagle11 @SteveRustad1 Yea he'll definitely get the rust belt,Florida,Michigan,and Pennsylvania....suuurrree… https://t.co/5p17g2U5Zt

user: enderton_justin
tweet: Renewable energy can generate billions of dollars in health benefits, study finds #RenewableEnergy #health via… https://t.co/9JoBDSHFn2

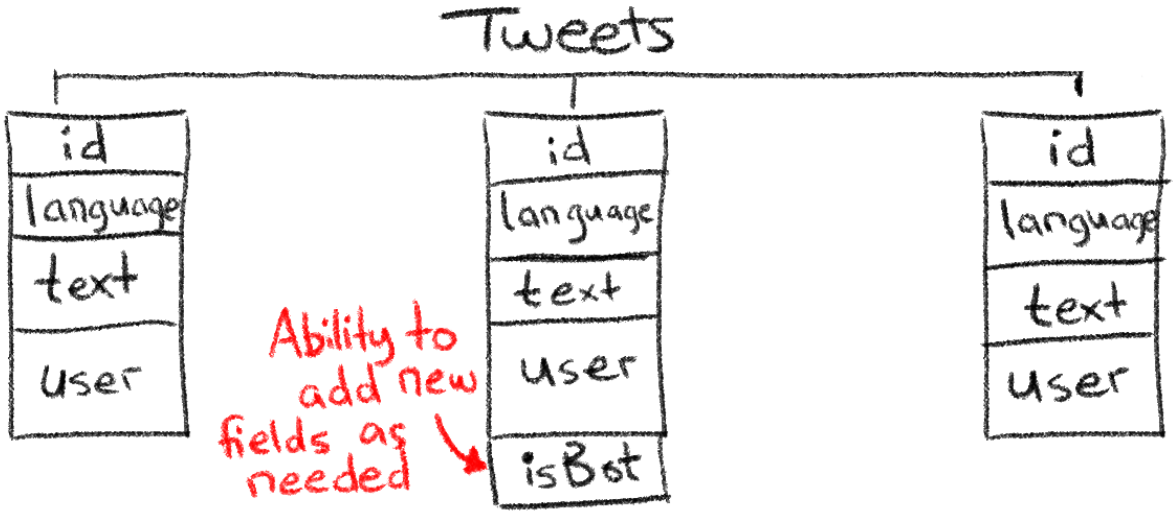
Part 1.2 - NoSQL Databases

What is a NoSQL Database?

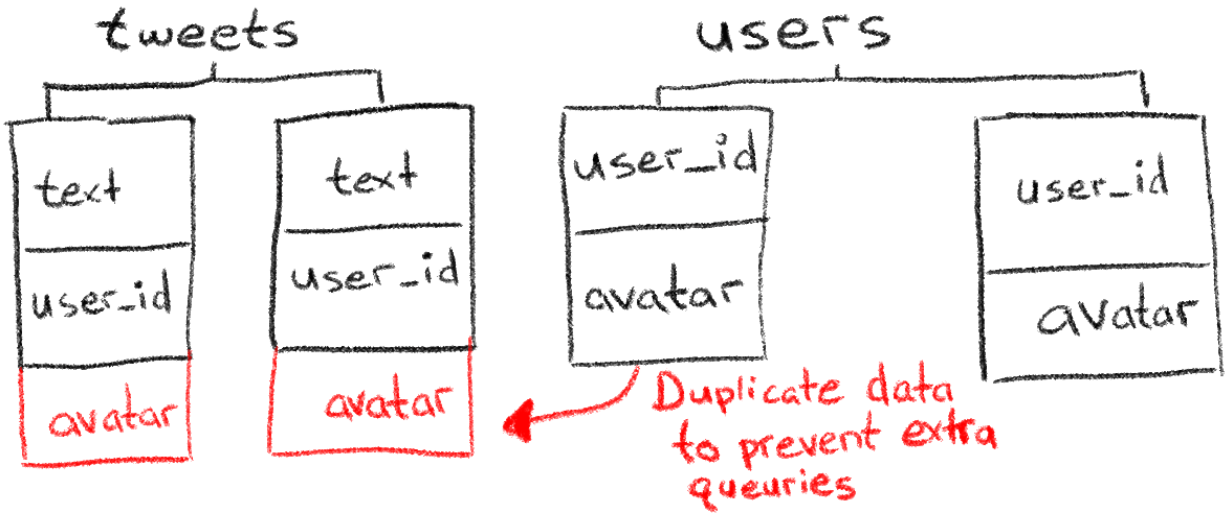
A NoSQL database is not as strictly defined as a SQL database. It is really any database that is not a SQL database. The key difference is that a SQL database finds data 'relationally' by performing qeueries that group shared columns together; where-as a NoSQL database generally needs a path or an id that points directly to the file in question. Data can still be stored relationally, in that a piece of data can hold an id pointing to another piece of data, but this type of query would take two seperate commands to get to that second piece of data.

Where a SQL database enforces strict typing and defining columns ahead of time, a NoSQL database is "schema-less" meaning that the type of data isn't enforced when it is stored. Furthermore, there is no consistent language used in a NoSQL database, such as the SQL language or CRUD operation workflow, rather, access to data is usually governed by a RESTful API (REpresentational State Transfer), a style for building systems that ensures consistent results.

A schema-less interface is convenient for a project like mine, where I might discover a new field that I need while working with the data, I don't need to modify every piece of data in the database to add the new field.



Not having SQL means that we have to get creative at times with how we store and retrieve data. For example, say I want to store all my tweets as well as the users who tweeted them. I probably want to have a seperate collection of users, and because we can no longer use SQL joins to collect users and tweets in one query, it's better to make duplicate data in the two collections. This is because it is generally better to optimize reading data than writing it.



One of the biggest benefits of a NoSQL database is the way it is able to scale. When we have a lot of users on a SQL database, we need to scale vertically, meaning, putting our database on bigger machines. With a NoSQL database, the DBMS can spread the data across multiple servers behind the scenes, providing faster data access for users; this is known as scaling horizontally.

Google Firestore Database

Google's Firestore, is a NoSQL database in the cloud. Firestore is a "document" database, meaning that each datapoint is not a table, rather a document in a collection. Additionally, each document may contain more collections within.

Perhaps the coolest part of Firestore is its realtime capabilities. I could make a website that uses JavaScript to display graphs depicting my study that "subscribes" to the data, and each time it is updated, my website will recieve the updates and immediately display the results without the need for another qeuery.

Firestore can be extended with "Cloud Functions" which run on the database at specified triggers such as new documents being added, I won't be exploring this right now.

Lets add our tweets to Firestore using the Python API.

In [257]:
```
from firebase import firebase_auth

# Get the db connection from firestore
db = firebase_auth.get_db()

# Get the tweets collection
tweets_collection = db.collection('tweets')

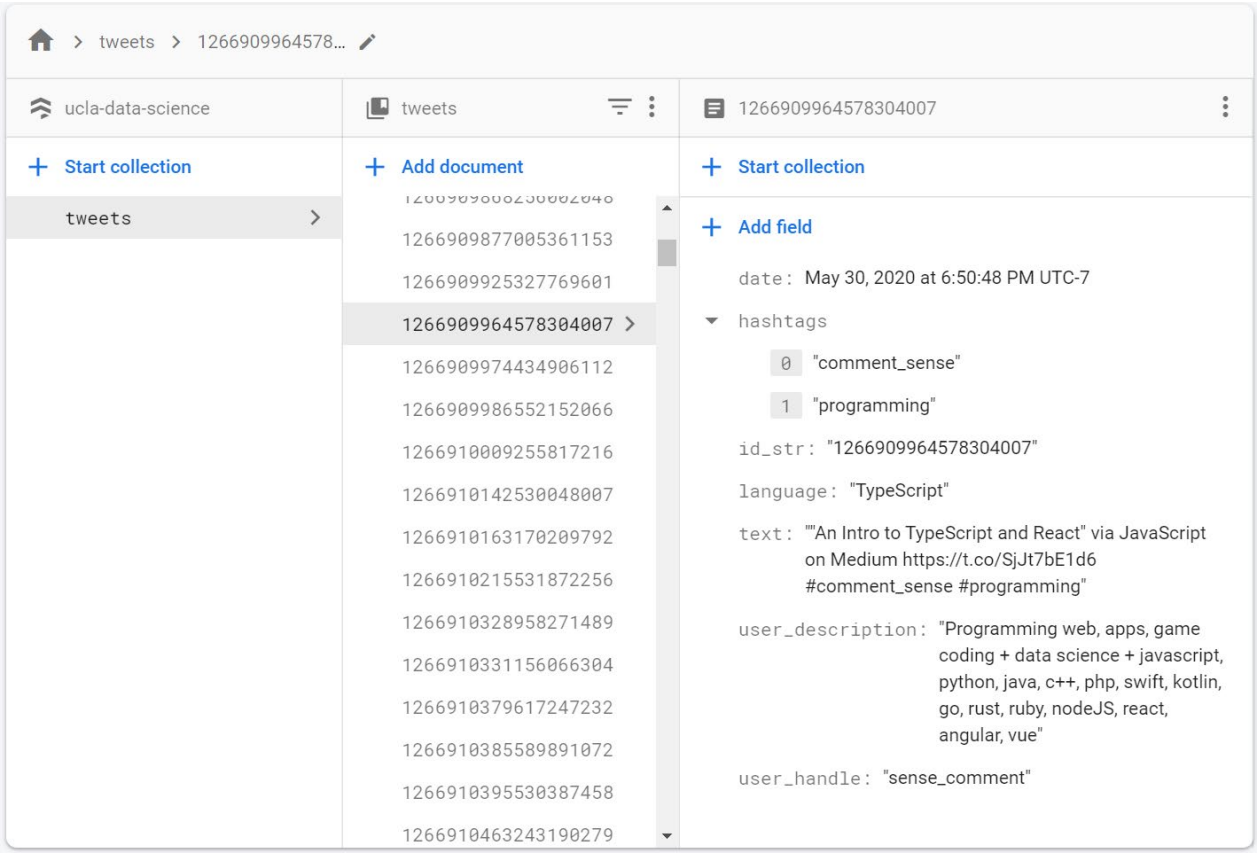
# Loop through all the tweets and add them to a batch, so we set them all
at once
```

```
batch = db.batch()
batch_size = 0
for language, tweets in tweets_dict.items():
    for _tweet in tweets:
        document_id = _tweet.id_str
        data = _tweet.__dict__
        data['language'] = language
        batch.set(tweets_collection.document(document_id), data)

        # Firestore limits batch writes to 500 items, so we will break the
 batch write up into groups of 500
        batch_size += 1
        if batch_size > 499:
            batch.commit()
            batch_size = 0

batch.commit()
```

Firestore has a web app we can use to browser or modify our data.



And, of course, we can retrieve our data from Firestore with a simple qeuery..

In [258]:
```
tweets_dict: DefaultDict[str, List[Tweet]] = defaultdict(list)
tweets_collection = db.collection('tweets').stream()
for doc in tweets_collection:
    _tweet = Tweet(None, doc.to_dict())
    tweets_dict[doc.get('language')].append(_tweet)

for k, v in tweets_dict.items():
    print(f'{k}: {len(v)}')
```

Python: 3801
Rust: 1037
TypeScript: 257
WebAssembly: 38
Kotlin: 156

Part 2: Algorithms

Supervised vs. Unsupervised Algorithms

As the names suggest, the difference between supervised and unsupervised algorithms is that supervised algorithms are observed and directed with additional data to get more accurate classifications for better future predictions. A supervised model uses a labelled dataset to classify outcomes. Classification is the process of labelling an item to be in a set of groups. Supervised algorithms can also be used for regression analysis, which is predicting a trend.

Unsupervised learning means that we do not supplement the model with correct output answers, rather we let it work on it's own to discover information that may not be visible to the human eye. The algorithms for unsupervised learning are generally more complicated as we do not know beforehand the outcomes to be expected. Unsupervised learning is used for clustering, which is the analysis of patterns and groupings from unlabeled data.

Reinforced Algorithms

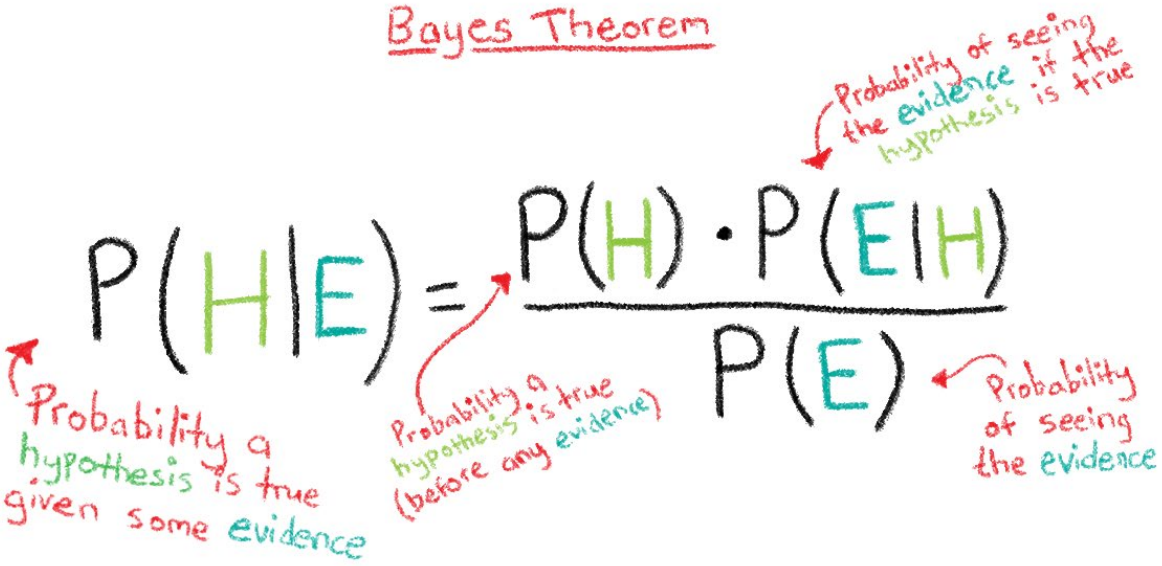
Reinforced algorithms are provided a signal indicating how well they did after they run to aid in future iterations.

Part 2.1: Naive Bayes (Supervised)

The Naive Bayes machine learning algorithm uses supervised learning to classify data. Historically, this algorithm has been used for text categorization, such as spam filters, which makes it the perfect algorithm to help us detect whether or not our tweets are actually about the programming language which they mention.

Naive Bayes is based on applying Bayes Theorem with independence assumptions between the features. Independence means that the occurance of one piece of evidence does not affect the probability of another piece of evidence. The key mantra behind Bayes Theorem is that new evidence does not completely determine your beliefs, rather it should update prior beliefs. In other words, seeing new evidence should restrict the space of possiblities.

A rule of thumb for where Bayes Theorem can be applied is when you have a hypothesis (this tweet is not about programming) and you have some evidence (the tweet contains the words "Python" and "snake") and you want to know the probability that your hypothesis holds given that the evidence is true.



If we let our hypotheses, H, be "this tweet is not about programming" and our evidence, E, be "the tweet contains the words 'President Trump'", Bayes Theorem tells us that the probability that the message is not about programming is the probability that ANY of our tweets is not about

programming AND contains "President Trump" divided by the probability that ALL of our tweets contain "President Trump".

We will need to have a vocabulary of many words which will indicate that a tweet is not about programming. Lets examine some tweets about the Python programming language..

In [264]:
```
import random

# Make a list of all the Python tweets
python_tweets: List[Tweet] = []
for doc in db.collection('tweets').where("language", "==", "Python").strea
m():
    _tweet = Tweet(None, doc.to_dict())
    # Don't add to our list of tweets if it is a duplicate
    if not any(x for x in python_tweets if x.text == _tweet.text):
        python_tweets.append(_tweet)

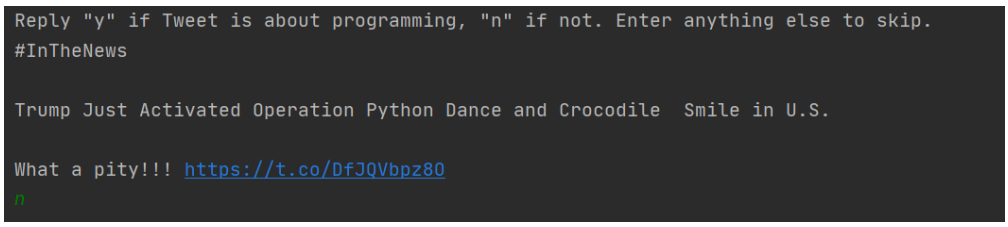
print(f'{len(python_tweets)} unique tweets containing "Python". Here\'s a
random sample of 5 of them: \n')

# Choose a random sample of 5 tweets and print them
for _tweet in random.sample(python_tweets, 5):
    print(f'~~~ Tweet: {_tweet.text}')
```

1823 unique tweets containing "Python". Here's a random sample of 5 of them:

~~~ Tweet: Writing Arabic to a CSV file using Python 3 https://t.co/W5YfPocL26
~~~ Tweet: Allow to define colors more flexibly https://t.co/2YJLXmvQec #github #Python #Dockerfile #Shell
~~~ Tweet: @FrancescoCiull4 I wanna build interactive apps and games, is it OK if I learn python(n.b I have a knowledge of bot… https://t.co/TQXgste7vD
~~~ Tweet: @Toblerone_DOTUS the period appropriate version of Monty Python and the holy grail. https://t.co/AhxeafU35G
~~~ Tweet: Python is my favorite program ##Python勉強中

While examining the Tweets, I can see that there will be many that are not about programming.
In order to apply Bayes Theorem to all of our Tweets, I will need to create a group of labelled data.
To do this, I created a Python program in the file "data_labeller.py", which shows me random tweets and lets me give an input 'y' or 'n', and then saves the results to a Firestore document in a new collection called "labelled_tweets". And then I labeled Tweets until I got bored..



(I don't know what this means, but I know it isn't about programming..)

In [199]:
```
from labeled_tweet import LabeledTweet

labeled_tweets: List[LabeledTweet] = []
for document in db.collection('labelled_tweets').stream():
    labeled_tweets.append(LabeledTweet(**document.to_dict()))
```
~~~

```
tweets_about_programming = [x for x in labeled_tweets if x.is_programming]

print(f'{len(labeled_tweets)} labeled tweets. {len(tweets_about_programmin
g)} are about programming.')
```

268 labeled tweets. 195 are about programming.

Now we need to make a function to tokenize incoming tweets into distinct lowecase words.

In [200]:
```
from typing import Set
import re

def tokenize(text: str) -> Set[str]:
    text = text.lower()
    all_words = re.findall("[a-z0-9']+", text)
    return set(all_words)
```

Next, we'll make a class for our Naive Bayes classifier. It will need to track tokens, counts, and labels from the training data. Our class will have a function "train" which accepts our labeled tweets and trains the class with them. We will also need a function "probabilites" to calculate the probability that a certain token is in a programming Tweet or not. Finally, we'll need a function "predict" which will handle the math equation of Bayes Theorem using our variables.

In [265]:
```
import math
from collections import defaultdict

class NaiveBayesClassifier:
    def __init__(self, smoothing: float = 0.5):
        self.smoothing = smoothing
        self.tokens: Set[str] = set()
        self.token_not_programming_counts: Dict[str, int] = defaultdict(in
t)
        self.token_programming_counts: Dict[str, int] = defaultdict(int)
        self.not_programming_messages = self.programming_messages = 0

    def train(self, _tweets: List[LabeledTweet]) -> None:
            for _tweet in _tweets:
                # Increment message counts
                if not _tweet.is_programming:
                    self.not_programming_messages += 1
                else:
                    self.programming_messages += 1

                # Increment word counts
                for token in tokenize(_tweet.text):
                    self.tokens.add(token)
                    if not _tweet.is_programming:
                        self.token_not_programming_counts[token] += 1
                    else:
                        self.token_programming_counts[token] += 1


    def _probabilities(self, token: str) -> Tuple[float, float]:
```

```
        """returns P(token | not_programming) and P(token | programming)""
"
        not_programming = self.token_not_programming_counts[token]
        programming = self.token_programming_counts[token]

        p_token_not_programming = (not_programming + self.smoothing) / (se
lf.not_programming_messages + 2 * self.smoothing)
        p_token_programming = (programming + self.smoothing) / (self.progr
amming_messages + 2 * self.smoothing)

        return p_token_not_programming, p_token_programming


def predict(self, text: str) -> float:
            text_tokens = tokenize(text)
            log_prob_if_not_programming = log_prob_if_programming = 0.0

            # Iterate through each word in our vocabulary
            for token in self.tokens:
                prob_if_not_programming, prob_if_programming = self._proba
bilities(token)

                # If token is found, add the log probability of seeing it
                if token in text_tokens:
                    log_prob_if_not_programming   +=   math.log(prob_if_not_pr
ogramming)
                    log_prob_if_programming += math.log(prob_if_programmin
g)

                # Otherwise add the log probability of not seeing it

                else:
not_programming)     log_prob_if_not_programming += math.log(1.0 - prob_if_
                    log_prob_if_programming += math.log(1.0 - prob_if_prog
ramming)

            prob_if_not_programming = math.exp(log_prob_if_not_programming
)
            prob_if_programming = math.exp(log_prob_if_programming)
            return prob_if_not_programming / (prob_if_not_programming + pr
ob_if_programming)
```

Now we have our classifier! Lets try it out on our collection of Python Tweets.

In [262]:
```
naiveBayesModel = NaiveBayesClassifier(smoothing=0.5)
naiveBayesModel.train(labeled_tweets)

from collections import Counter

predictions = [{'tweet': x.text, 'prediction': naiveBayesModel.predict(x.t
ext)} for x in python_tweets]
predictions.sort(key=lambda x: x['prediction'], reverse=True)

not_programming_tweets = [x for x in predictions if x['prediction'] > 0.5]

print(f'Out of {len(python_tweets)} Tweets, found {len(not_programming_twe
```

```
ets)} that weren\'t related to programming.\n')
print('Here are the top 20 Tweets furthest away from programming:')
for prediction in predictions[:10]:
    print(f'~~~ Tweet: {prediction["tweet"]}\n')

print('\n\nHere are the top 10 Tweets that were deffinitely about programm
ing:')
for prediction in predictions[-10:]:
    print(f'~~~ Tweet: {prediction["tweet"]}\n')
```

Out of 1823 Tweets, found 207 that weren't related to programming.

Here are the top 10 Tweets furthest away from programming:

~~~ Tweet: Here’s is this Monty Python scene where a bonch of tertiary foods just fighting each other in the wa r in a kitchen… https://t.co/s7rK5bg241

~~~ Tweet: @paddy12725650 My favourite Monty Python me and my brother still chuck random quotes at eac h other and have made my kids watch it too

~~~ Tweet: RT @NiisBBB: Made my first snake/python/cobra mix adopt and she is sassy >=U! Can say she loves a nice firm hug here and there. Hiss hiss…

~~~ Tweet: Worshippers on Zoom: is anyone else reminded of this scene from Monty Python’s Meaning of Lif e, when we greet each… https://t.co/OkncllaBAi

~~~ Tweet: @scottEweinberg when i was growing up we only had the audio tapes of monty python, faulty towe rs and other classics… https://t.co/d3BI5bw3qH

~~~ Tweet: I actually enjoy taikas work a lot (skipped the na zi comedy, sorry ol chap) but this comment is for a monty python boxed set.

~~~ Tweet: RT @nur_mx1: "you're flicking the nose of a python, which is fine, but you've gotta be fairly sure h ow to deal with the python when it stri…

~~~ Tweet: RT @J0nasF: @Pythika A bird family that - surprisingly - evolved in Scandinavia(!)
(So Monty Python's "Norwegian blue" wasn't completely of…

~~~ Tweet: Yeah walked the walk (albeit Monty Python style) and certainly can’t talk the talk (but he’s good at spaffing and b… https://t.co/aCuHbywBGc

~~~ Tweet: It blows my mind that, in the film Sliding Doors, a man who quotes Monty Python in the pub is uni ronically presente… https://t.co/Q27cYaxTOe

Here are the top 10 Tweets that were deffinitely about programming:

~~~ Tweet: RT @gp_pulipaka: Springer has Released 65 #DataScience #Books for Free. #BigData #Analytics #IoT #IIoT #Python #RStats #JavaScript #ReactJS…

~~~ Tweet: RT @gp_pulipaka: Free Book: Gaussian Processes for #MachineLearning . #BigData #Analytics # DataScience #AI #IoT #IIoT #Python #RStats#JavaS…

~~~ Tweet: RT @gp_pulipaka: 20 Best #DataScience #Books for Beginner &amp; Experts in 2020. #BigData # Analytics #IoT #IIoT #Python… https://t.co/jvog9vc8MP
~~~

~~~ Tweet: 6 Free Data Science Books. #BigData #Analytics #Hadoop #DataScience #NLProc #IoT #IIoT #Python #RStats #JavaScript… https://t.co/PZDtC8ACDi

~~~ Tweet: RT @IoTMLBigData: 6 Free Data Science Books. #BigData #Analytics #AI #MachineLearning #Hadoop #DataScience #NLProc #IoT #IIoT #Python #RSta…

~~~ Tweet: RT @gp_pulipaka: Basics of Machine Learning Algorithm. #BigData #Analytics #DataScience #AI #MachineLearning #IoT #IIoT #Python #RStats #Ja…

~~~ Tweet: RT @gp_pulipaka: 6 Free Data Science Books. #BigData #Analytics #Hadoop #DataScience #NLProc #IoT #IIoT #Python #RStats #JavaScript #ReactJ…

~~~ Tweet: 6 Free Data Science Books. #BigData #Analytics #AI #MachineLearning #Hadoop #DataScience #NLProc #IoT #IIoT #Python… https://t.co/Afl96O7juJ

~~~ Tweet: RT @gp_pulipaka: 6 Free Data Science Books. #BigData #Analytics #AI #MachineLearning #Hadoop #DataScience #NLProc #IoT #IIoT #Python #RStat…

And there we have it!

Our data is properly clasified thanks to the Naive Bayes algorithm! As you can see, it was classified with pretty terrific accuracy.

It's pretty easy to see why the most-likely-to-be-programming Tweets were categorized as such; each one is just a list of programming related hashtags.

Examing the list of Tweets that were not about programming, we can see that they mostly discuss Monty Python or pythons the animal. We can actually examine the tokens that make up the least likely programming Tweets to see what words appear the most.

In [255]:
```
def probability_given_token(token: str, model: NaiveBayesClassifier) -> float:
    prob_if_not_programming, prob_if_programming = model._probabilities(token)
    return prob_if_not_programming / (prob_if_not_programming + prob_if_programming)

words = sorted(naiveBayesModel.tokens, key=lambda t: probability_given_token(t, naiveBayesModel))

print("non_programming_words", words[-10:])
print("programming_words", words[:10])
```

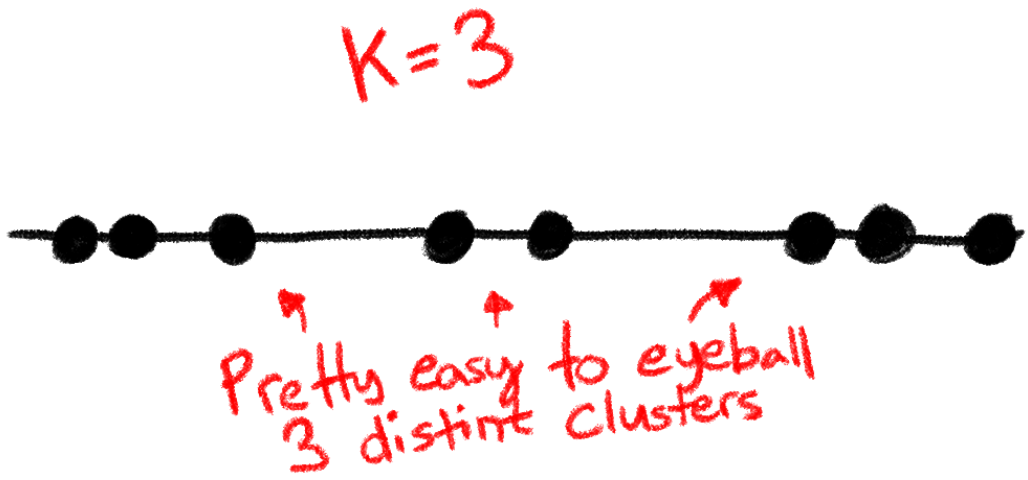
non_programming_words ['b', 'meaning', 'each', 'operation', 'other', 'grail', 'ball', 'snake', 'holy', 'monty']
programming_words ['learning', 'learn', 'data', 'machine', 'programming', 'code', '100daysofcode', 'using', 'free', '2020']

Part 2.2 - K-Means Clustering

Our Naive Bayes algorithm depended on labeled data to predict the outcome of unlabeled data. Clustering, however, is an example of machine learning in which we work with unlabeled data only. When examing a dataset, clusters of values can often appear. For example, looking at voters, their preferred political party will often cluster by region. K-means is one of the simplest clustering methods.

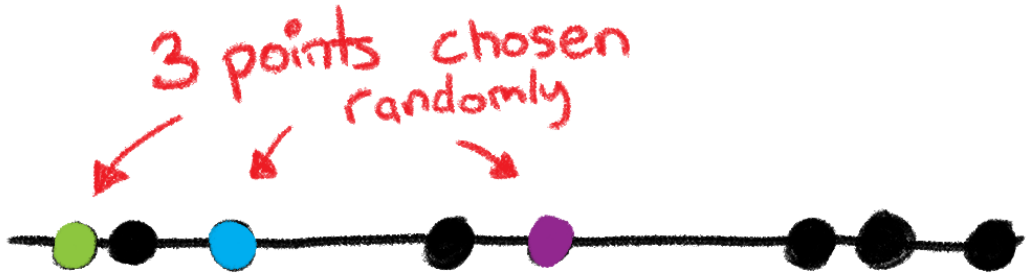
Step 1: The K

The K in K-means represents the number of clusters to make, it is chosen in advance, either manually or by another algorithm. In the following simple example, I'll choose 3.



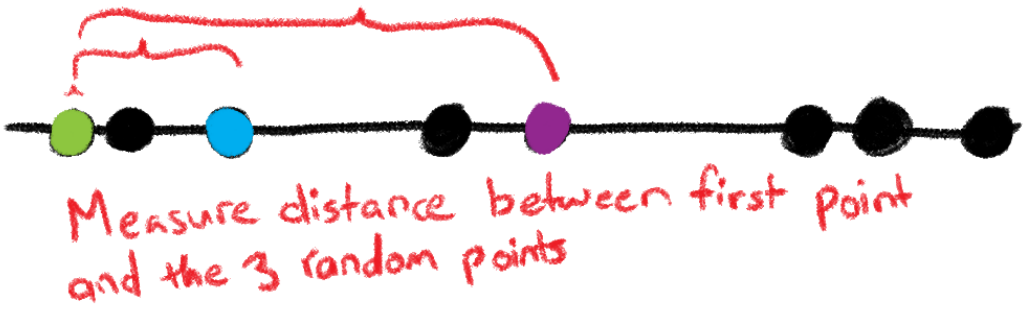
Step 2: Random

Randomly select 3 distinct data points. These are the initial clusters.



Step 3: Measure

Measure the distance between the first point, and the 3 initial clusters. In our example, the first randomly selected point just happens to land on the very first point, so the first measurement is 0.

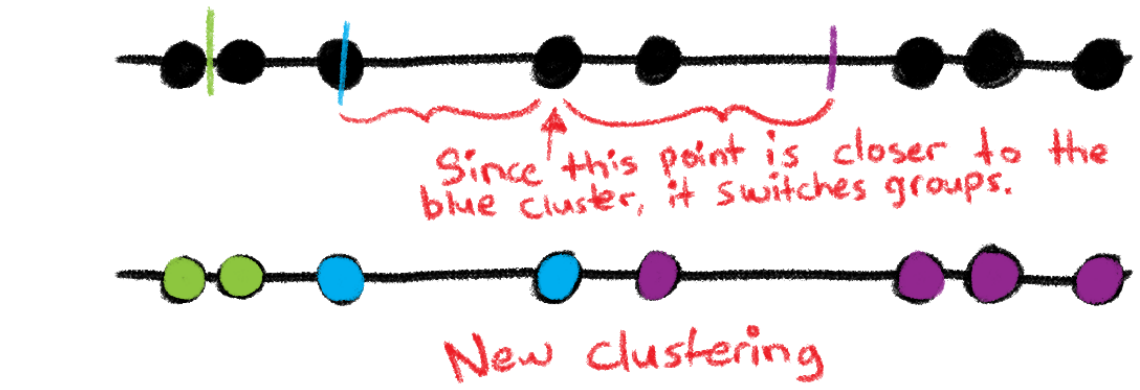


Step 4: Assign Initial Clusters

Assign the points to the clusters closest to them by measuring the distances starting from the first point.

Step 5: Calculate the Mean of Each Cluster

Calculate the mean of each cluster, and then re-assign the points based on the points closest to each mean.

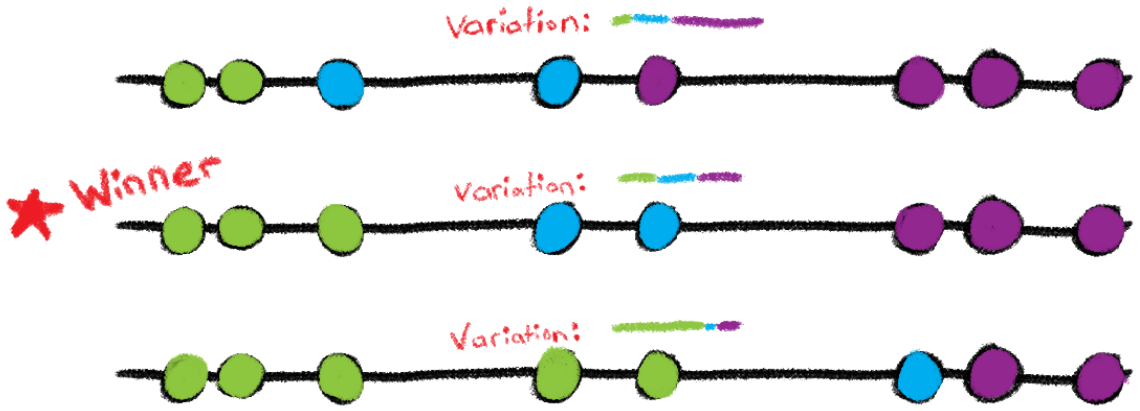


Step 5: Assess New Clusters

As you can see, this is not the optimal clustering, k-means clustering is an iterative process that needs to run multiple times to find the best clustering fit. Add up the total variation between each cluster, so it can be measured against future iterations.

Step 6: Start Over

Now that we have our first iteration in the bag, we need to start from step 1 with new randomly selected points. The resulting maximum variation will be compared to the maximum variation of the first run-through and this will be repeated until we feel we have enough attempts. The iteration with the smallest variation within the clusters will be chosen as the best clustering.



K-Means Clustering - Real World Example

Lets say that we want to create a series of online live-Tweeting programming session for Python users. Now that we have collected several days worth of Tweets, and each one has a timestamp, we can see which parts of the day will see the most programmers on Twitter. Thanks to K-Means clustering, we can not only find the most active part of the day, but we can also find multiple times of day with the most traffic, just by changing the 'K' value. For instance, we may find that prime Tweeting time is found in groups at 8am, 12pm, and 7pm.