Challenges

Description

Automatic summarization is the process of reducing a text document with a computer program in order to create a summary that retains the most important points of the original document. A number of algorithms have been developed, with the simplest being one that parses the text, finds the most unique (or important) words, and then finds a sentence or two that contains the most number of the most important words discovered. This is sometimes called “extraction-based summarization” because you are extracting a sentence that conveys the summary of the text. For your challenge, you should write an implementation of a text summarizer that can take a block of text (e.g. a paragraph) and emit a one or two sentence summarization of it. You can use a stop word list (words that appear in English that don’t add any value) from here.

Challenge Input

> At GitHub we recently revamped how we do DNS from the ground up. This included both how we interact with external DNS providers and how we serve records internally to our hosts. To do this, we had to design and build a new DNS infrastructure that could scale with GitHub’s growth and across many data centers. Previously GitHub’s DNS infrastructure was fairly simple and straightforward. It included a local, forwarding only DNS cache on every server and a pair of hosts that acted as both caches and authorities used by all these hosts. These hosts were available both on the internal network as well as public internet. We configured zone stubs in the caching daemon to direct queries locally rather than recurse on the internet. We also had NS records set up at our DNS providers that pointed specific internal zones to the public IPs of this pair of hosts for queries external to our network. This configuration worked for many years but was not without its downsides. Many applications are highly sensitive to resolving DNS queries and any performance or availability issues we ran into would cause queuing and degraded performance at best and customer impacting outages at worst. Configuration and code changes can cause large unexpected changes in query rates. As such scaling beyond these two hosts became an issue. Due to the network configuration of these hosts we would just need to keep adding IPs and hosts which has its own problems. While attempting to fire fight and remediate these issues, the old system made it difficult to identify causes due to a lack of metrics and visibility. In many cases we resorted to tcpdump to identify traffic and queries in question. Another issue was running on public DNS servers we run the risk of leaking internal network information. As a result we decided to build something better and began to identify our requirements for the new system. We set out to design a new DNS infrastructure that would improve the aforementioned operational issues including scaling and visibility, as well as introducing some additional requirements. We wanted to continue to run our public DNS zones via external DNS providers so whatever system we build needed to be vendor agnostic. Additionally, we wanted this system to be capable of serving

> both our internal and external zones, meaning internal zones were only available on our internal network unless specifically configured otherwise and external zones are resolvable without leaving our internal network. We wanted the new DNS architecture to allow both a deploy-based workflow for making changes as well as API access to our records for automated changes via our inventory and provisioning systems. The new system could not have any external dependencies, too much relies on DNS functioning for it to get caught in a cascading failure. This includes connectivity to other data centers and DNS services that may reside there. Our old system mixed the use of caches and authorities on the same host, we wanted to move to a tiered design with isolated roles. Lastly, we wanted a system that could support many data center environments whether it be EC2 or bare metal.