Design Fake News Detector

Q: How are we defining fake news?

A: Let's define fake news as news articles that contain intentionally misleading info from what appears to be a reliable source.

Q: Who are we designing the system for?

A: Social media platforms. They're our primary customers.

Q: Why are these social media platforms paying us?

A: Our service aims to limit the negative financial, health, and social impacts of misinformation on the platforms.

Q: For clarification, is this a reasonable way that one of these platforms would use our service: they call our service with a news article that's been posted on their platform, and depending on our response, they have logic around allowing the post, limiting exposure of the post, or disallowing the post?

A: Yes, that sounds reasonable, but our service doesn't tell the platform how they should handle unreliable news articles. In other words, you shouldn't be concerned with that.

Q: What do our clients expect us to provide to them through our service?

A: Our platform returns a response of either 'reliable' or 'unreliable' depending on the provided news article.

Q: I'm assuming that we need to be able to explain why our service returns 'reliable' or 'unreliable'. Is this reasonable?

A: Yes, being able to explain our decision is important to both the model development and our clients if they have an inquiry.

Q: How will these social media platforms interact with our service?

A: Our service is API-based.

Q: Do clients call our service with the raw text of articles?

A: They can, but they can also just send us the URL of a news article.

Q: Do we have to implement a way to extract articles from provided URLs, or can we assume that we're provided with that functionality?

A: Yes, we have to implement that extraction.

Q: Do clients expect higher latencies if they send us just the URL of an article?

A: Yes, and we can also reject requests when we can't extract an article from the provided URL.

Q: Do we allow clients to bring their own models or data for our system to operate with?

A: No, we only use our proprietary models and data.

Q: Do we need to build an ML platform supporting dozens of scientists to rapidly iterate our model development?

A: No, I'm more concerned with how your system handles data and hosts a single model.

1. Gathering System Requirements

As with any ML design interview question, the first thing that we want to do is to gather requirements; we need to figure out what we need to build and what we don't need to build.

We're designing a system which detects fake news for social media platforms.

The system should provide each of the following functionalities:

- Scrape news articles and fake news datasets from the web
- Wrangle data from various sources into a unified structure
- Labelling interface to enable a workforce to label newly scraped articles
- Extract linguistic features from the scraped articles
- Model support including training, validation, hyper-parameter tuning, and hosting
- Online model experimentation
- Extensive Logging

We'll need to rely on systems which we won't touch on or design:

- Authentication and authorization of our system's API
- Cloud infrastructure provider

2. Web Scraper

We need to scrape news articles from the web. This involves handling web pages with:

- HTTPS
- Javascript
- Login prompts

- Paywalls
- CAPTCHAs

The Web Scraper can be created from scratch with libraries such as Beautiful Soup but that's not recommended. We can instead use 3rd party services like scrapestack or the dedicated article scraper plugin for ScrapeBox. However, there's also open source web scraping libraries available such as Scrapy. Most of these tools will cover our requirements and we'll be able to avoid creating our own custom web scraper. The Web Scaper will be responsible for polling the top 100 news sites in the US (CNN, New York Times, Huffington Post, etc). We need to establish contractual agreements with news sources that don't allow web scraping. The newly scraped articles will be sent to the Data Wrangler.

For the fake news datasets, we can download them directly from their sources (with permission for commercial use) including the Information Security and Object Technology (ISOT) Fake News dataset as well as the Kaggle Fake News dataset. These datasets can be placed in an object store (such as S3) which will be periodically polled by the Data Wrangler to be processed.

3. Data Wrangler

Because we're using multiple data sources, we need to wrangle the data so that the data is structured in a consistent way for creating features. This includes:

- Date and time format manipulations
- Changing data types to match a consistent schema
- Replacing missing data with NAs
- Joining data
- Exporting data to a storage layer

These functionalities can be developed from scratch using Pyspark libraries on a Spark cluster. As well, Spark allows us to perform stream processing so we don't have to temporarily store the raw data before we wrangle it. After processing, the articles will be stored. Our storage layer will be a PostgreSQL database where each row will be keyed on an article ID. The rows will contain columns including the article contents and article metadata such as the publication date, publisher, article length, and perhaps the news article author. We'll partition the database tables on the article publication date so we can use date ranges to create data sets to train on. After the Data Wrangler has finished, the articles need to be labeled as either reliable or unreliable.

4. Labelling Interface

We'll need to implement a way for a workforce to label the newly scraped articles as either reliable or unreliable. Labelling must be done manually with either an internal workforce or we can outsource the labelling task to a 3rd party service such as Mechanical Turk. Comprehensive label guidelines will have to be defined so that articles are correctly and consistently labeled. An interface will need to be built to display news articles to the labelers along with the labelling guidelines and buttons to allow labelers to label an article as reliable or unreliable. Architecturally, when the Data Wrangler adds an unlabeled article to the database we can have a Change Data Capture mechanism on the database (for PostgreSQL the write-ahead logs are used to track changes to a table) which will add a task to a queue (AWS SQS). When labelers log in to the Labelling Interface, the queue will be polled for an unlabeled article and displayed to the labeler for labelling. The interface can be a simple web app hosted on a lightweight framework such as Flask. The Flask server can directly add labels to the label column in the same PostgreSQL table used by the Data Wrangler for each article.

5. Feature Extractor

In addition to labelling potentially hundreds of thousands of news articles, we need to extract linguistic features from them so that we can train our model.

Our primary source of features will be generated with Linguistic Inquiry and Word Count (LIWC pronounced 'Luke'). We can also use additional features such as the news article publisher or whether or not the article is featured on a site that requires HTTPS.

LIWC extracts over 90 features for a provided text. It works by calculating to which degree various categories of features are observed within the given text including:

- Summary Language Variables (words per sentence, tone, words with more than 6 letters)
- Linguistic Dimensions (total pronouns, prepositions, conjugations)
- Psychological Processes (positive/negative emotions, social references, informal language)

LIWC generates features by maintaining dictionaries, or lexicons, pertaining to each feature category and counting the occurrences of words in each respective dictionary. Numerical features are then calculated to represent the percentage of the words which were allocated to each category. For instance, if there are 5 conjugations within a 100 word text then the conjugation feature would be 0.05 (or 5%).

Architecturally, when the Data Wrangler adds an article to the database we can have a Change Data Capture component on the database which will add a task to a queue (here RabbitMQ). A cluster of Celery workers will poll tasks from the queue and perform the LIWC feature

extraction. Since the lexicons of LIWC are proprietary, we'll need to integrate with either the Receptiviti API or work to get a custom solution of a local LIWC library so we don't have to make an API call for each news article. When a Celery worker is finished extracting features from an article, the features are added to the feature store (here FEAST). FEAST has the ability to generate statistics about the features such as number of missing features, minimum/max/mean/mode/standard deviation of feature values, and also the number of unique values across features. This will allow the system to produce data quality metrics which we can create alarms around so we can readily discover anomalies and data drift. The statistics generated by FEAST are fully compatible with Tensorflow Data Validation (TFDV) which provides libraries to identify anomalies and data drift. The logging and alarms will be implemented in the same way that we implement logging in the Extensive Logging section (LogStash, Elasticsearch, and Kibana).

6. Model Support

In terms of our model, we'll need to support:

- Training
- Tuning
- Evaluation
- Hosting
- Explainability

For model training, we'll need to split a labeled feature set into a test set and a training set. The training set will be used to train and tune the model (with a subset assigned as the validation set for tuning). The test set will be used to perform offline model evaluation. We'll be using a random forest as our model. We'll start with 128 trees each with a max depth of six. This will make the model small enough to fit in a single machine and the model training should take under 1 hour to train on modern dual core processors which means we don't necessarily need implement model training checkpoints. The average news article length is under 800 words and the average word length in English is 4.7 characters. A UTF-8 character requires at most 4 bytes of memory which means we can assume each article is on average under 15 kilobytes (KB). This means a dataset of two hundred thousand news articles will likely take up 3 gigabytes (GB) of memory. This data can fit in the RAM of commodity hardware which means we don't need to partition our data set into chunks for incremental training. This means we don't have to implement an online learning process. Since the proportion of news articles that are unreliable is so few compared to those that are reliable, we'll need to account for this in our training process so we avoid degraded model performance. There's commonly two techniques used in

the case of random forests. The first approach to mitigating unbalanced data sets is to use class weighting. To implement class weighting, we simply add a stronger weight for the minority class when calculating the Gini impurity for a split - this usually results in a higher rate of false positives. The second approach (and a more appropriate method for our use case) is to use Synthetic Minority Oversampling Technique (SMOTE). SMOTE generates synthetic examples within the minority class to be used for training. It does this by randomly selecting a minority example and drawing connecting lines between it and its nearest minority neighbors in the feature space. It then generates minority examples by creating examples which lie on the connecting lines. SMOTE generally produces less false positives than class weighting but models generally benefit from separately trying both methods and picking whichever one performs better.

For model tuning we'll need to take a subset of the training set and create a validation set. This validation set will be used to tune the hyperparameters of the random forest model. These hyperparameters can include max tree depth, number of trees, max number of leaf nodes, minimum impurity decrease for a split, minimum samples in a leaf, minimum samples required to perform a split, and the max number of features to evaluate at each split. To search for the optimal hyperparameters in an efficient way, we'll use bayesian hyperparameter tuning. After tuning is complete, we need to serialize the model and save it to a model store for hosting or for later reference.

Evaluating the performance of our random forest model should incorporate metrics in terms of the business as well as the model itself. For business metrics, we can measure the number of complaints clients reach out to us about and perhaps the number of defamation suits filed against news articles our system deems reliable. For model metrics, we need to be careful that we don't rely heavily on binary accuracy. Since our data set is unbalanced, accuracy can present a deceiving degree of high performance. Better metrics like the F1 score and perhaps Cohen's kappa should be used. Cohen's kappa calculates the model performance in comparison to a model which randomly predicts reliable or unreliable in direct proportion to the class imbalance within the dataset. Finally, when evaluating our model, it's important to recognize how much the model helps identify fake news compared to not having the model. To determine this, we can use Bayes' theorem. Assuming that the prior of fake news is 2% and that our model's true positive rate is 92% and that the false positive rate is 3%, we get a 38% probability of a news article being unreliable given that our model predicts an article as unreliable. Compared to the 18% probability of an article being unreliable given that a human predicts it to be unreliable, our model is roughly twice as good as a human. We can appropriately say that this is a useful model.

In terms of model explainability, we'll use Shapley values. Shapley values estimate the impact that each individual feature has on the overall prediction from our model. This way, when we're asked for an explanation, we can look at the Shapley values of the model and the feature values for the sample in question and see how strongly each feature influenced the prediction.

For model hosting, we'll need to load the tuned model into the hosting service on startup from the model store (S3). As well, since clients are able to send request with just URLs, we'll need to call on the Web Scraper, the Data Wrangler, and the Feature Extractor in order to obtain features from the URL. This means APIs need to be created around these services. Architecturally, we can use a production Flask server to host our API. A cluster of these servers will sit behind a load balancer and be equipped with an NGINX reverse proxy which will send requests to a Gunicorn WSGI which forwards the requests to our Flask server. As well, we'll also need monitoring around our model to detect model drift. Model drift occurs when articles that were at one time correctly classified are now being incorrectly classified due to the changing nature of how news articles evolve. The monitoring and alarms will be implemented in the same way that we implement logging in the Extensive Logging section (LogStash, Elasticsearch, and Kibana). Calling the APIs for the Web Scraper, the Data Wrangler and the Feature Extractor could add significant latency for the client when they provide only a URL.

Finally, we'll want to implement a mechanism (guardrails) to override the model behavior in the case that we know things that the model does not. For instance, The Onion is a popular satirical news publisher. With guardrails we can return to the client that articles published by The Onion should not be taken down and therefore should be deemed reliable.

7. Online Model Experimentation

Even though we're only expected to build a single model, we still need to support online model experimentation so we can build our confidence when replacing existing logic. We can build this element ourselves from scratch. This should include p-value calculations for A/B tests or expected improvement calculations for multi-armed bandits. These calculations should be based on a relevant business metric such as client complaints and at least one model metric such as the F1 score. These calculations can be done once we learn the true label of each article in the experiment. For A/B testing, each article within a request will be assigned to a particular treatment of either the model or the existing logic. To avoid assigning the same article to more than one treatment - which can happen in the case that the same article gets sent to our service more than once - we'll need to generate a unique identifier for each article content. In this case, we can use a checksum. Each request from the client will be accompanied by a request ID which will be used to eventually associate online predictions with true labels for the prediction. This will enable us to perform online evaluations of the model which can be used to compare against the existing logic. True labels will be generated either by a client complaint or by the guidelines

established in the Labelling Interface. This means that each article requested by our clients will also have to be enqueued in the Labelling Interface. We can accomplish by using the same Change Data Capture used by Labelling Service except we'll now need to monitor the request ID table. Architecturally, we will create another table in the PostgreSQL database where each row is keyed on the request ID. Each row will have columns indicating what our model predicted as well as what the true label is eventually determined to be.

8. Extensive Logging

The request ID associated with each client request can also be used as a reference for future inquiries. This request ID will be logged in our system such that if a client wants to know more details about a specific response returned by our system, then we can look up that request ID. This request ID will be associated with the specifics of the requests (article, timestamp, guardrails used, experiment treatment) as well as other information to help us debug. This will include the model parameters, hyperparameters, the feature set used to train the model (as well as pointers to the respective unfeatured dataset), the validation and evaluation performances, and also the Shapley values associated with the model/article combination of that particular request. Some of this information will also need to be added to the request ID database table for future truth labelling - so if a write to the database fails, we need to fail the request and ask the client to retry. In this case, we'll also log that the request failed to complete. Architecturally, the logging will be done on the Flask server using the default logging library. Logs will be sent to LogStash which will asynchronously write logs to Elasticsearch which will allow us to quickly query the logs. Kibana will be used on top of Elasticsearch to allow our developers to view logs with a web browser. To be alerted as soon as possible when something is seriously wrong (for instance always returning 'unreliable' to client), we can setup alarms with Kibana and connect the alarms to services like PagerDuty to alert the relevant teams (DevOps or MLOps). We'll also need to create a custom Kibana dashboard to allow Technical Account Managers (TAMs, typically the primary point of contact for enterprise clients) to quickly search request IDs to investigate and find explanations (Shapley values) for specific prediction responses returned by our system.

9. System Diagram