Of course. Here is a detailed, line-by-line explanation of the provided R code, organized by the project's phases and key topics.

R Programming Project: Nepali News Analytics

This project performs a comprehensive analysis of a Nepali news dataset. The workflow includes loading data, cleaning it, performing Exploratory Data Analysis (EDA), classifying news articles into categories using machine learning, conducting sentiment analysis, summarizing articles, and performing Named Entity Recognition (NER).

Phase 0: Setup and Library Loading

This section loads all the necessary R packages for the project. Each package provides a set of functions for specific tasks like data manipulation, text mining, machine learning, and visualization.

Code Block 1: Core Libraries

```
library(readr); library(dplyr); library(stringr); library(tm); library(quanteda); library(qu
```

Explanation: * `readr`: For fast and efficient reading of rectangular data like CSV files (`read_csv`). * `dplyr`: A powerful package for data manipulation and transformation (e.g., `filter`, `select`, `group_by`, `summarise`). * `stringr`: Provides a consistent and easy-to-use set of functions for working with strings and text. * `tm`: The "Text Mining" package, one of the original frameworks for NLP in R. Although `quanteda` is more modern and used more heavily here, `tm` might be used for specific functions or is a dependency. * `quanteda` & `quanteda.textmodels`: A modern and powerful framework for quantitative text analysis. It's used for creating a corpus, tokenizing text, building document-feature matrices (DFM), and training text-specific models like Naive Bayes. * `SparseM`: A package for handling sparse matrices, which are common in text analysis where most words don't appear in most documents. * `doParallel`: Used to enable parallel processing, which can significantly speed up computationally intensive tasks by using multiple CPU cores. * `ranger`: A fast implementation of the Random Forest algorithm, which is used for classification.

Code Block 2: Additional Machine Learning & Visualization Libraries

```
library(Matrix); library(pheatmap); library(reshape2); library(textclean); library(glmnet);
```

Explanation: * `Matrix`: Provides classes and methods for dense and sparse matrices. * `pheatmap`: For generating pretty heatmaps, often used to visualize confusion matrices or feature correlations. * `reshape2`: Used for restructuring data, particularly with the `melt` function to transform data from a "wide" to a "long" format, which is ideal for `ggplot2`. * `textclean`: Contains tools for cleaning and preprocessing text data. * `glmnet`: Implements fast algorithms for fitting generalized linear models with regularization, used here for sentiment analysis. * `text2vec`: Another high-performance text analysis framework, often used for word embeddings and topic modeling. Here, it's used for tokenization and creating a DTM for sentiment analysis. * `tidytext`: Implements a "tidy" approach to text mining, integrating well with `dplyr` and other tidyverse packages. * `ggplot2`: The premier package for data visualization in R, used to create all the plots in this project. * `wordcloud`: For generating word cloud visualizations.

Code Block 3: Final Set of Libraries

```
library(RColorBrewer); library(caret); library(e1071); library(randomForest); library(udpipe
```

Explanation: * `RColorBrewer`: Provides color palettes for creating more visually appealing plots. * `caret`: (Classification And REgression Training) A comprehensive framework for building and evaluating machine learning models. It simplifies tasks like data splitting, pre-processing, and model tuning. * `e1071`: Contains functions for various machine learning algorithms, including Support Vector Machines (SVM), which is used for classification. * `randomForest`: The original implementation of the Random Forest algorithm. `ranger` (loaded earlier) is a faster alternative, but this might be loaded for comparison or as a dependency. * `udpipe`: A package for natural language processing that performs tokenization, part-of-speech tagging, lemmatization, and dependency parsing. It's used here for Named Entity Recognition. * `textrank`: An algorithm for text summarization, used to extract the most important sentences from an article.

Phase 1: Data Collection and Preprocessing

This phase focuses on loading the dataset and cleaning it to make it suitable for analysis.

Code Block 1: Loading the Dataset

```
news_data<- read_csv("50k_news_dataset.csv",show_col_types = FALSE)
```

Explanation: * `read_csv()`: This function from the `readr` package reads the `50k_news_dataset.csv` file into a data frame (specifically, a tibble) named `news_data`. * `show_col_types = FALSE`: This argument suppresses the message that would normally print the data type for each column, keeping the console output clean.

Code Block 2: Initial Data Exploration

```
str(news_data)
head(news_data)
summary(news_data)
dim(news_data)
```

Explanation: * `str(news_data)`: Displays the **str**ucture of the `news_data` object, showing the column names, their data types (`chr` for character), and the first few values. This is a quick way to get an overview of the dataset. * `head(news_data)`: Shows the first 6 rows of the dataset. * `summary(news_data)`: Provides a statistical summary for each column. For character columns, it gives the length, class, and mode. * `dim(news_data)`: Returns the **dim**ensions of the dataset (number of rows, number of columns). Initially, it's 50,000 rows and 4 columns.

Code Block 3: Checking and Removing Missing Values

```
colSums(is.na(news_data))
news_data<-na.omit(news_data)
dim(news_data)
```

Explanation: * `is.na(news_data)`: Creates a logical matrix of the same size as `news_data`, with `TRUE` where a value is missing (`NA`) and `FALSE` otherwise. * `colSums(...)`: Calculates the sum of `TRUE` values for each column, effectively counting the number of missing values in each column. The output shows 3 missing values in the `content` column. * `na.omit(news_data)`: Creates a new data frame by removing any rows that contain at least one `NA` value. The result is assigned back to `news_data`. * `dim(news_data)`: Shows the new dimensions. The row count has decreased from 50,000 to 49,997, confirming that 3 rows were removed.

Code Block 4: Checking and Removing Duplicate Rows

```
sum(duplicated(news_data))
news_data[duplicated(news_data), ]
news_data<-news_data[!duplicated(news_data), ]
dim(news_data)
```

Explanation: * `duplicated(news_data)`: Returns a logical vector where `TRUE` indicates that a row is an exact duplicate of a row that appeared earlier in the data frame. * `sum(...)`: Summing this logical vector counts the total number of duplicate rows (which is 5). * `news_data[duplicated(news_data), ]`: This uses logical indexing to display the actual duplicate rows that were found. * `!duplicated(news_data)`: The `!` negates the logical vector, so `TRUE` now represents unique rows. * `news_data[...]`: This filters the data frame, keeping only the unique rows. The result is assigned back to `news_data`. * `dim(news_data)`: Shows the final dimensions after removing duplicates. The data now has 49,992 rows.

Code Block 5: Text Pre-processing Pipeline

```
clean_nepali_text<-function(text) {
  text<-as.character(text)
  text<-gsub("<.*?>","",text)
  text<-gsub("http\\S+|www\\S+|https\\S+","",text)
  text<-gsub("\\S+@\\S+","",text)
  text<-gsub("\\s+"," ",text)
  text<-trimws(text)
  return(text)
}

news_data$content<-clean_nepali_text(news_data$content)
news_data$heading<-clean_nepali_text(news_data$heading)
```

Explanation: * `clean_nepali_text <- function(text) { ... }`: Defines a custom function to clean the Nepali text. * `as.character(text)`: Ensures the input is treated as a character string. * `gsub("<.*?>", "", text)`: Uses `gsub` (global substitution) to find and remove any HTML tags (e.g., `<p>`, `<a>`). The pattern `<.*?>` matches anything between `<` and `>`. * `gsub("http\\S+|www\\S+|https\\S+", "", text)`: Removes URLs. `\\S+` matches one or more non-whitespace characters. * `gsub("\\S+@\\S+", "", text)`: Removes email addresses. * `gsub("\\s+", " ", text)`: Replaces one or more whitespace characters (`\\s+`) with a single space. This cleans up extra spaces, tabs, or newlines. * `trimws(text)`: Removes any leading or trailing whitespace from the text. * The function is then applied to both the `content` and `heading` columns of the `news_data` data frame, overwriting them with the cleaned text.

Code Block 6: Creating a `quanteda` Corpus

```
news_corpus<-corpus(news_data, text_field = "content")
docvars(news_corpus, "category")<-news_data$category
docvars(news_corpus, "source")<-news_data$source
```

Explanation: * `corpus(...)`: This `quanteda` function converts the `news_data` data frame into a `corpus` object. A corpus is a specialized data structure for managing and analyzing a collection of texts. * `text_field = "content"`: Specifies that the `content` column contains the main text of the documents. * `docvars(news_corpus, ...)`: `docvars` are document-level variables (metadata). This code attaches the `category` and `source` from the original data frame as metadata to each corresponding document in the `news_corpus`. This is crucial for later analysis, like classifying articles by category.

Code Block 7: Tokenization and Document-Feature Matrix (DFM) Creation

```
news_tokens<-tokens(news_corpus,
                    remove_punct = TRUE,
                    remove_symbols = TRUE,
                    remove_numbers = TRUE,
                    remove_url = TRUE)
news_dfm<-dfm(news_tokens)
news_dfm<-dfm_trim(news_dfm,min_docfreq = 0.01, docfreq_type = "prop")
```

Explanation: * `tokens(news_corpus, ...)`: This function tokenizes the text, which means breaking it down into individual words (tokens). The arguments remove punctuation, symbols, numbers, and URLs during this process. * `dfm(news_tokens)`: This creates a **D**ocument-**F**eature **M**atrix from the tokens. A DFM is a large matrix where rows represent documents, columns represent unique words (features), and the values are the counts of how many times each word appears in each document. * `dfm_trim(news_dfm, ...)`: This function trims the DFM to reduce its size and remove noise. * `min_docfreq = 0.01`: This parameter specifies the minimum document frequency for a word to be kept. * `docfreq_type = "prop"`: This means the frequency is interpreted as a **prop**ortion. So, this line removes any word that does not appear in at least 1% of all documents. This is a common feature reduction technique to remove very rare words.

Phase 2: Exploratory Data Analysis (EDA)

This phase explores the dataset to understand its characteristics, such as the distribution of news sources, categories, and article lengths.

Code Block 1: Analyzing Category and Source Distributions

```r
source_counts <- count(news_data, source, sort = TRUE)
category_counts <- count(news_data, category, sort = TRUE)
```

Explanation: * `count(news_data, source, sort = TRUE)`: This `dplyr` function counts the number of occurrences of each unique value in the `source` column. `sort = TRUE` arranges the results in descending order. The same logic applies to the `category` column. This gives a quick look at which news sources and categories are most common.

Code Block 2: Analyzing Heading and Content Lengths

```r
heading_character_length <- nchar(news_data$heading)
summary(heading_character_length)

content_character_length <- nchar(news_data$content)
summary(content_character_length)

heading_word_count <- str_count(news_data$heading, "\\w+")
summary(heading_word_count)

content_word_count <- str_count(news_data$content, "\\w+")
summary(content_word_count)
```

Explanation: * `nchar(...)`: This base R function calculates the number of **char**acters in each string. It's used here to find the character length of each heading and content. * `str_count(..., "\\w+")`: This `stringr` function counts the number of matches to a pattern. The pattern `\\w+` matches one or more "word characters" (letters, numbers, underscore), effectively counting the number of words. * `summary(...)`: This function is called on the resulting numeric vectors to get descriptive statistics (Min, Median, Mean, Max, etc.) for the lengths.

Code Block 3: Top Terms Analysis

```
top_features<-topfeatures(news_dfm,50)
top_features_df<-data.frame(
  Term=names(top_features),
  Frequency=as.numeric(top_features)
)
```

Explanation: * `topfeatures(news_dfm, 50)`: This `quanteda` function extracts the 50 most frequent terms (features) from the `news_dfm`. It returns a named numeric vector. * `data.frame(...)`: This code converts the named vector into a more usable data frame with two columns: `Term` and `Frequency`.

Code Block 4: Word Cloud Generation

```
set.seed(123)
wordcloud(words = top_features_df$Term[1:50],
          freq = top_features_df$Frequency[1:50],
          min.freq = 2,
          max.words = 100,
          random.order = FALSE,
          rot.per = 0.35,
          colors = brewer.pal(8, "Dark2"))
```

Explanation: * `set.seed(123)`: Sets a seed for the random number generator. This ensures that the word cloud layout is the same every time the code is run, making it reproducible. * `wordcloud(...)`: This function from the `wordcloud` package generates the visualization. * `words`, `freq`: Specify the terms and their corresponding frequencies. * `max.words = 100`: Sets the maximum number of words to display in the cloud. * `random.order = FALSE`: Plots the most frequent words in the center. * `rot.per = 0.35`: Sets the proportion (35%) of words that will be rotated vertically. * `colors`: Specifies the color palette to use for the words.

Code Block 5: N-gram Analysis (Bigrams and Trigrams)

```
news_bigrams<-tokens_ngrams(news_tokens,n=2)
bigram_dfm<-dfm(news_bigrams)
top_bigrams<-topfeatures(bigram_dfm, 28)

news_trigrams<-tokens_ngrams(news_tokens,n=3)
trigram_dfm<-dfm(news_trigrams)
top_trigrams<-topfeatures(trigram_dfm,20)
```

Explanation: * `tokens_ngrams(news_tokens, n=2)`: This function takes the single-word tokens and combines them into sequential pairs (bigrams). For example, "the quick brown fox" becomes "the_quick", "quick_brown", "brown_fox". `n=3` creates trigrams (three-word sequences). * The rest of the code follows the same pattern as the single-word analysis: a DFM is created from the n-grams, and `topfeatures` is used to find the most common ones.

Code Block 6: Lexical Diversity Analysis (Type-Token Ratio)

```r
calculate_lexical_diversity <- function(tokens) {
  types <- length(unique(unlist(tokens)))
  tokens_total <- length(unlist(tokens))
  return(types / tokens_total)
}

lexical_diversity <- news_data %>%
  group_by(category) %>%
  summarise(
    avg_lexical_diversity = mean(sapply(strsplit(content, "\\s+"),
      function(x) length(unique(x)) / length(x))), .groups = "drop")
```

Explanation: * `calculate_lexical_diversity`: This custom function calculates the Type-Token Ratio (TTR), a measure of lexical diversity. "Types" are the unique words, and "tokens" are the total number of words. A higher TTR means a wider variety of vocabulary is used. * `news_data %>% ...`: This code uses a `dplyr` pipeline to calculate the average TTR for each news category. * `group_by(category)`: Groups the data frame by the `category` column. * `summarise(...)`: Calculates a summary statistic for each group. * `sapply(strsplit(content, "\\s+"), ...)`: For each article's content, this splits it into words (`strsplit`), then applies an anonymous function to calculate the TTR (`length(unique(x)) / length(x)`). * `mean(...)`: Calculates the average of these TTRs for all articles within a category.

Phase 3: News Category Classification

This is the core machine learning phase, where models are trained to predict the category of a news article based on its content.

Code Block 1: Stratified Sampling and Data Splitting

```r
set.seed(123)
stratified_indices <- createDataPartition(news_data$category,
                                          p = 0.4, # 40% of 50K = 20K
                                          list = FALSE)
news_data_20k <- news_data[stratified_indices, ]

trainIndex <- createDataPartition(news_data_20k$category, p = 0.8, list = FALSE)
```

```r
train_data <- news_data_20k[trainIndex, ]
test_data <- news_data_20k[-trainIndex, ]
```

Explanation: * `set.seed(123)`: Ensures the data partitioning is reproducible. * `createDataPartition(...)`: A `caret` function for creating balanced splits of the data. * The first `createDataPartition` call creates a **stratified sample** of 40% of the original data. Stratification ensures that the proportion of articles in each category in the 20k subset is the same as in the original 50k dataset. This is crucial for building a representative model. * The second call splits this 20k subset into training (80%) and testing (20%) sets, again using stratification.

Code Block 2: Preparing Data for Modeling (DFM and TF-IDF)

```r
train_corpus <- corpus(train_data, text_field = "content")
# ... (similar for test_corpus, train_tokens, test_tokens, train_dfm, test_dfm)

train_dfm <- dfm_trim(train_dfm, min_docfreq = 0.01, docfreq_type = "prop")
test_dfm <- dfm_match(test_dfm, features = featnames(train_dfm))

train_tfidf <- dfm_tfidf(train_dfm)
test_tfidf <- dfm_tfidf(test_dfm)
```

Explanation: * This section repeats the corpus creation, tokenization, and DFM creation steps, but now separately for the training and test sets. * `dfm_match(test_dfm, ...)`: This is a critical step. It forces the `test_dfm` to have the exact same set of features (columns) as the `train_dfm`. This is necessary because a trained model expects a specific set of features in a specific order. * `dfm_tfidf(...)`: This function converts the raw word counts in the DFM to **T**erm **F**requency-**I**nverse **D**ocument **F**requency (TF-IDF) weights. * **TF (Term Frequency):** How often a word appears in a document. * **IDF (Inverse Document Frequency):** Gives more weight to words that are rare across all documents and less weight to common words (like "the", "is"). * TF-IDF is a powerful weighting scheme that often improves model performance by highlighting the words that are most uniquely descriptive of a particular document.

Code Block 3: Model 1 - Naive Bayes Classifier

```r
nb_model <- textmodel_nb(train_tfidf, train_data$category, smooth = 1)
nb_predictions <- predict(nb_model, newdata = test_tfidf)
nb_confusion <- confusionMatrix(nb_predictions, as.factor(test_data$category))
```

Explanation: * `textmodel_nb(...)`: This `quanteda.textmodels` function trains a Naive Bayes classifier. It takes the TF-IDF weighted training DFM (`train_tfidf`) and the corresponding labels (`train_data$category`) as input. `smooth = 1` is Laplace smoothing to prevent issues with words that don't appear in a certain category. * `predict(...)`: Uses the trained `nb_model` to make predictions on the unseen test data (`test_tfidf`). * `confusionMatrix(...)`: This `caret` function compares the model's predictions with the true labels from the test data to generate a detailed evaluation, including accuracy, precision, recall, F1-score, and a confusion matrix.

Code Block 4: Model 2 - Support Vector Machine (SVM)

```
svm_model <- svm(train_sparse, train_data$category,
                 kernel = "linear", cost = 1)
svm_predictions <- predict(svm_model, test_sparse)
svm_confusion <- confusionMatrix(svm_predictions, as.factor(test_data$category))
```

Explanation: * `svm(...)`: This function from the `e1071` package trains a Support Vector Machine. * `train_sparse`: The TF-IDF matrix is converted to a sparse matrix format, which is more efficient for algorithms like SVM. * `kernel = "linear"`: Specifies a linear kernel, which is a good baseline for text classification problems with many features. * `cost = 1`: A regularization parameter that controls the trade-off between a smooth decision boundary and classifying training points correctly. * The `predict` and `confusionMatrix` steps are analogous to the Naive Bayes model.

Code Block 5: Model 3 - Random Forest

```
rf_model <- ranger(category ~ .,
                   data = train_features,
                   num.trees = 50,
                   num.threads = parallel::detectCores() - 1,
                   verbose = TRUE,
                   seed = 123)
rf_predictions <- predict(rf_model, test_features)$predictions
rf_confusion <- confusionMatrix(rf_predictions, test_features$category)
```

Explanation: * `ranger(...)`: This function trains a high-performance Random Forest model. * `category ~ .`: This is a formula interface, meaning "predict `category` using all other columns (`.`) as features." * `data = train_features`: The input data is a standard data frame where the TF-IDF matrix has been converted. * `num.trees = 50`: Builds an "ensemble" of 50 decision trees. * `num.threads`: Uses parallel processing to speed up training.

Code Block 6: Model Comparison and Performance Visualization

```
model_comparison <- data.frame(
  Model = c("Naive Bayes", "SVM", "Random Forest"),
  Accuracy = c(
    round(nb_confusion$overall['Accuracy'], 4),
    #... and so on for other models
  ),
  #... and so on for Precision, Recall, F1_Score, Kappa
)
```

Explanation: * This code block aggregates the key performance metrics (Accuracy, Precision, Recall, etc.) from the three confusion matrix objects (nb_confusion, svm_confusion, rf_confusion) into a single, clean data frame called model_comparison. This makes it easy to directly compare the performance of the different algorithms.

Code Block 7: Saving Results and Models

```
saveRDS(nb_model, "models/naive_bayes_20k.rds")
# ... similar for other models
ggsave("outputs/model_performance_comparison.png", performance_plot, ...)
classification_summary <- list(...)
saveRDS(classification_summary, "outputs/classification_summary_20k.rds")
```

Explanation: * saveRDS(...): Serializes and saves an R object to a file. This is used to save the trained models (nb_model, svm_model, etc.) so they can be reloaded later without needing to be retrained. It is also used to save the final summary list. * ggsave(...): Saves a ggplot plot to a file (e.g., as a PNG). * classification_summary <- list(...): Creates a comprehensive list object that neatly stores all relevant information about the classification task: dataset info, model performance, feature info, and processing details. This is excellent practice for documenting and reproducing results.

Phase 4: Sentiment Analysis

This section builds a model to classify text as positive, negative, or neutral. It uses a different, pre-labeled dataset for training.

Code Block 1: Loading and Preparing Sentiment Data

```
data <- read_csv("nepali_sentiment_lexicon.csv")
# ... text cleaning steps ...
split <- createDataPartition(data$Sentiment, p = 0.8, list = FALSE)
train <- data[split, ]
test <- data[-split, ]
```

Explanation: * A new dataset, nepali_sentiment_lexicon.csv, is loaded. This dataset contains sentences labeled with a sentiment score. * The text is cleaned using functions from the textclean package (tolower, replace_non_ascii, etc.). * The data is split into 80% training and 20% testing sets using createDataPartition.

Code Block 2: Feature Engineering and Model Training (glmnet)

```
it_train <- itoken(train$clean_text)
vocab <- create_vocabulary(it_train)
vectorizer <- vocab_vectorizer(vocab)
dtm_train <- create_dtm(it_train, vectorizer)
model <- cv.glmnet(x = dtm_train, y = train$Sentiment, family = "multinomial", type.measure
```

Explanation: * This section uses the `text2vec` package workflow. * `itoken`: Creates an iterator over the text to process it memory-efficiently. * `create_vocabulary`: Builds a vocabulary of unique words from the training data. * `vocab_vectorizer`: Creates a vectorizer object based on the vocabulary. * `create_dtm`: Uses the vectorizer to create a DTM from the text. * `cv.glmnet(...)`: Trains a regularized logistic regression model. * `family = "multinomial"`: Specifies that there are more than two outcome classes (e.g., positive, negative, neutral). * `cv.glmnet` automatically performs **c**ross-**v**alidation to find the optimal regularization parameter (`lambda`).

Code Block 3: Prediction and Evaluation

```
pred <- predict(model, dtm_test, s = "lambda.min", type = "class")
confusionMatrix(as.factor(pred), as.factor(test$Sentiment))
```

Explanation: * `predict(...)`: Makes predictions on the test set's DTM. * `s = "lambda.min"`: Uses the optimal value of lambda found during cross-validation. * `type = "class"`: Asks the model to output the predicted class label directly. * `confusionMatrix`: Evaluates the performance of the sentiment model.

Code Block 4: Creating a Prediction Function

```
predict_sentiment <- function(text, model, vectorizer_obj) {
  # ... (clean text, create dtm)
  predictions <- predict(model, dtm_new, s = "lambda.min", type = "class")
  probabilities <- predict(model, dtm_new, s = "lambda.min", type = "response")
  confidence <- apply(probabilities, 1, max)
  return(data.frame(...))
}
```

Explanation: * This code wraps the entire prediction process into a reusable function. It takes new text as input, cleans it, creates a DTM using the *same vectorizer from training*, predicts the sentiment class, calculates the model's confidence (the highest probability among the classes), and returns a neat data frame with the results. This is a very good practice for deploying a model.

Phase 5: Text Summarization with TextRank

This section implements a summarizer to extract the most important sentences from a news article.

Code Block 1: The Summarization Function

```
summarize_with_textrank <- function(text, n_sentences = 3) {
  sentences <- unlist(strsplit(text, "\\.|\\!|\\?"))
  #... (clean up sentences)

  #... (create a terminology data frame for textrank)
  tr <- textrank_sentences(data = sentences_df, terminology = terminology)
  top_sentences <- summary(tr, n = n_sentences, keep.sentence.order = TRUE)

  return(paste(top_sentences, collapse = ". "))
}
```

Explanation: * The function takes an article's text and the desired number of summary sentences (`n_sentences`) as input. * `strsplit`: Splits the text into individual sentences based on punctuation marks. * The code then prepares the data in the format required by the `textrank_sentences` function, which involves creating a data frame of sentences and a corresponding data frame of the words within each sentence. * `textrank_sentences(...)`: This is the core of the algorithm. It builds a graph where sentences are nodes and the similarity between sentences defines the edges. It then runs an algorithm similar to Google's PageRank to find the most "important" sentences (nodes). * `summary(tr, ...)`: Extracts the top `n` sentences from the TextRank object. * `paste(...)`: Joins the selected sentences back together into a single summary paragraph.

Code Block 2: Applying the Summarizer

```
for (i in sample_indices) {
  tryCatch({
    news_data$summary[i] <- summarize_with_textrank(news_data$content[i], 3)
  }, error = function(e) {
    #... (handle errors)
  })
}
```

Explanation: * This code loops through a sample of articles from the dataset. * For each article, it calls the `summarize_with_textrank` function to generate a 3-sentence summary. * `tryCatch({...})`: This is an error-handling block. If the summarization function fails for any reason (e.g., on a very short or unusual article), the `error` part is executed, preventing the entire script from crashing.

Phase 6: Named Entity Recognition (NER)

This final phase identifies named entities like people, places, and organizations in the text.

Code Block 1: Downloading and Loading the NER Model

```
model_download <- udpipe_download_model(language = "hindi", model_dir = "models/")
hindi_model <- udpipe_load_model(model_download$file_model)
```

Explanation: * `udpipe_download_model(language = "hindi")`: Downloads a pre-trained `udpipe` model. Since a pre-trained model for Nepali is not readily available in the package, a Hindi model is used as a proxy due to the linguistic similarities (shared script and vocabulary). * `udpipe_load_model(...)`: Loads the downloaded model into the R session so it can be used for annotation.

Code Block 2: NER Extraction Function

```r
extract_named_entities <- function(text, model) {
  result <- udpipe_annotate(model, x = text)
  result_df <- as.data.frame(result)
  entities <- result_df[result_df$upos == "PROPN", "token"]
  return(unique(entities[entities != ""]))
}
```

Explanation: * `udpipe_annotate(model, x = text)`: This is the main function that processes the text. It performs tokenization, part-of-speech (POS) tagging, and more, returning a detailed annotation. * The result is converted to a data frame. * `result_df[result_df$upos == "PROPN", "token"]`: This is the key step for NER. It filters the results to keep only the tokens that have been tagged with the Universal Part-of-Speech (`upos`) tag of `"PROPN"`, which stands for **Prop**er **N**oun. This is a common method for extracting named entities. * `unique(...)`: Returns only the unique entities found in the text.

Code Block 3: Applying NER to the Dataset

```r
all_entities <- c()
for(i in sample_articles) {
  entities <- extract_named_entities(news_data_20k$content[i], hindi_model)
  all_entities <- c(all_entities, entities)
  # ... (progress update)
}
entity_counts <- table(all_entities)
```

Explanation: * This code loops through a sample of 300 articles. * In each iteration, it calls the `extract_named_entities` function and appends the found entities to a master list called `all_entities`. * `table(all_entities)`: After processing all sample articles, this function counts the occurrences of each unique named entity, allowing the analysts to see which entities are mentioned most frequently.