Build a Brand Sentiment Analysis Extension for Twitter

Estimated time needed: 2 hours

Background

Are you a Twitter enthusiast who wants to quickly see the sentiment of the tweets on your timeline? Or maybe you're an organization and want to understand the sentiment behind your brand? Whatever the reason, this lab will show you how to create a simple browser extension that performs sentiment analysis on Twitter posts and displays the results in a pie chart using Chart.js.

Learning Objectives

By the end of this guided project, you will:

- Learn about browser extensions and their purpose.
- Explore the concept of sentiment analysis and its applications.
- Use a pre-trained natural language processing (NLP) model to analyze text data for sentiment.
- Use Chart.js to create and display a pie chart in a browser extension.
- Test and debug your browser extension to ensure it functions properly.
- (Optional) Create and deploy your browser extension to the add-on/extension market.

Introduction

In this project, you will be provided with a file directory containing empty files. You will then go through each file and fill it with the necessary code and information. As you progress through the project, the concepts and reasoning behind each step will be explained, allowing you to understand what is happening and why. This project is designed to be a hands-on learning experience, where you will build a functional tool while gaining a deeper understanding of the underlying concepts. The ultimate goal is to empower you to continue improving and expanding upon the project on your own.

What is a Browser Extension?

A browser extension is a useful tool that can be installed in a web browser and adds additional functionality to the browser. For example, a browser extension could be used to block ads, add a spell checker, or perform other tasks.

By using a browser extension, this lab makes it simple for users to access and analyze tweets directly from their web browser, eliminating the need to switch between applications.

What is Sentiment Analysis?

Sentiment analysis is a method of analyzing text data to determine the emotional tone or sentiment of the text.

Brand sentiment analysis is the process of analyzing public opinion about a brand by examining the sentiments expressed in social media posts, such as tweets. This can help companies understand how customers feel about their products or services, and identify areas for improvement.

What is Natural Language Processing?

Natural language processing (NLP) is the study of how computers can understand and manipulate human language.

NLP algorithms and models are used to analyze and interpret natural language data, such as text or speech, and extract meaningful information from it. This allows computers to automatically understand and analyze human language, enabling them to perform tasks such as language translation, sentiment analysis, and text summarization.

What is Chart.js?

Chart.js is a popular JavaScript library for creating charts and graphs. It's easy to use and provides a wide range of chart types, including pie charts, bar charts, and line graphs.

Step 1: Set Up the Extension

Please download the extension outline from this GitHub repository.

You can use the following commands in the terminal to do so:

```
git clone https://github.com/arora-r/twitter-sentiment-analysis-extension-outline.git
mv twitter-sentiment-analysis-extension-outline twitter-sentiment-analysis-extension
cd twitter-sentiment-analysis-extension
```

These commands download the files, rename the directory and then moves into that directory.

The only file thats been filled out is the `manifest.json`. This file defines the basic information about an extension, such as its name, version, and description. It also tells the browser that the extension has a browser action (a button that appears in the toolbar) and specifies the popup thats shown after the button in the toolbar is clicked.

Let's examine a few important fields.

Open **manifest.json** in IDE

Manifest Version

The `manifest version` is the version the extension will use and it is used to ensure compatibility between the extension and the browser. This lab will use version 2, which enables special features without having to do any extra configuration.

Permissions

The `permissions` attribute is an array of strings, where each string specifies a permission that the extension requires to function correctly. These permissions can include access to specific URLs, tabs, or other resources that the extension needs in order to work.

Content Scripts

The `content_scripts` are scripts that are injected into web pages by a browser extension to modify or add functionality to the page. They are specified in the manifest file of the extension under the content_scripts property, which is an array of objects that define the content scripts to be injected. For this lab, there only needs 1 object which has the following properties:

- `matches`: This is an array of strings that specify the URLs of the pages where the content script will be injected. The strings can use wildcards and regular expressions to match multiple pages.
- `js`: This is an array of strings that specify the JavaScript files to be injected into the page. The JavaScript files can be local to the extension or external URLs.

Browser Action

The `browser_action` property is used to define the appearance and behavior of the extension's button in the browser's toolbar, as well as control the popup thats shown for the extension (the front-end user interface for the extension). This lab's browser_action property is an object that has the following properties:

- `default_title`: This is a string that specifies the default title for the button, which is displayed as a tooltip when the user hovers over the button.
- `default_popup`: This is a string that specifies location of the .html file to be shown when the user clicks on the button.

Background

The `background` property is used to define the background script of the extension, which is a script that runs in the background of the browser and performs long-running tasks or functions that are not associated with a specific page. This lab's background property is an object that has the following properties:

- `scripts`: This is an array of strings that specify the JavaScript files that make up the background script. The files can be local to the extension or external URLs.

With the necessary directories set up and the relevant information reviewed, the next step is to access and analyze the tweets on the page.

Step 2: Accessing and Analyzing Tweets

For the purposes of this lab, the `tweet-sentiment.js` file will need to include functions that allow it to access the content of webpages. Specifically, the script will need to be able to collect all the tweets on the screen and analyze the sentiment of each tweet. Since content scripts have direct access to web page content, the following section contains the functions for the `tweet-sentiment.js` file.

Access the Tweets

Accessing the tweets is fairly simple, its just a simple query within Javascript.

```
// Look for the elements with the `data-testid="tweetText` attribute
tweets = document.querySelectorAll('[data-testid="tweetText"]');
```

The above is the Javascript syntax to look for all the elements on the page with the specific attribute: `data-testid="tweetText"`. This is possible because this attribute is seen on all text part of tweets, which is what the sentiment analysis will run on.

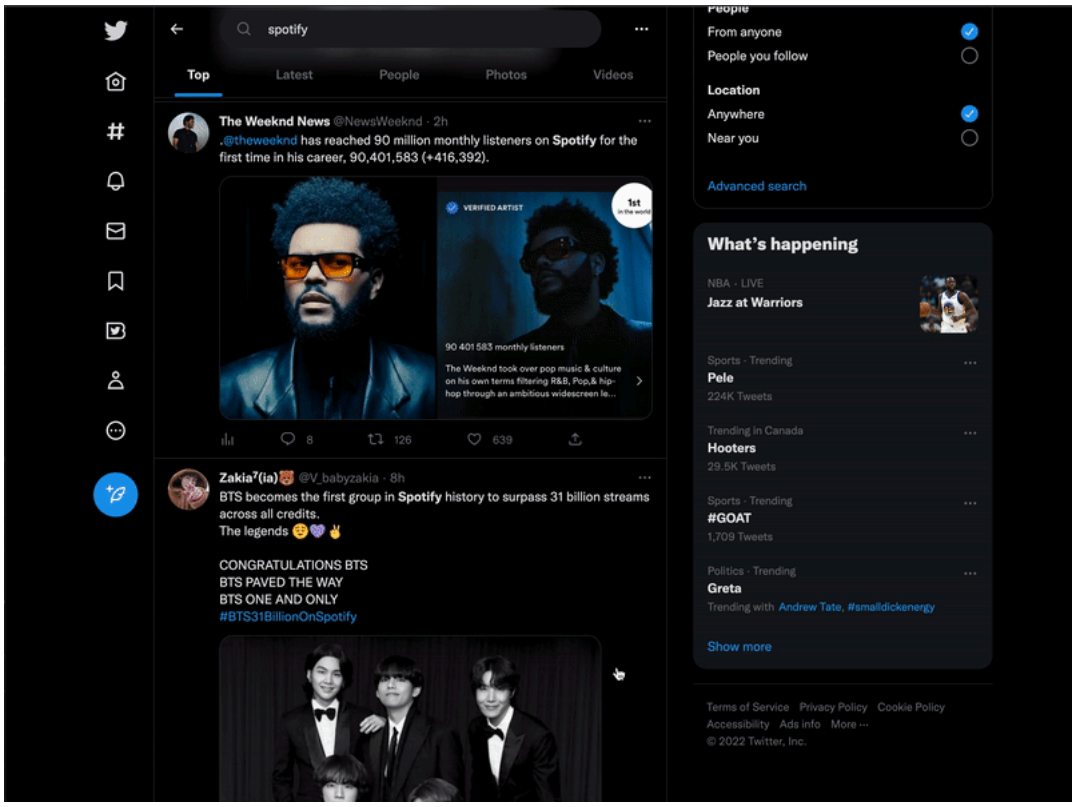
Inspect Element

For reference, to get the `data-testid="tweetText"` attribute, you can follow these steps to inspect the elements.

1. Right-click on the element you want to inspect on the webpage.
2. From the context menu that appears, select the "Inspect" or "Inspect Element" option. This will open the developer tools panel in your browser.
3. The developer tools panel will show you the HTML code for the page on the left, and the corresponding rendered page on the right. The element you selected will be highlighted in both the HTML code and the rendered page.
4. You can click on the element in the HTML code to view its properties in the right-hand panel. These properties include the element's tag name, attributes, and CSS styles.

In this case, the wanted element is the text in a tweet. So, right click on the text in a tweet, and click on the "Inspect" or "Inspect Element" option. This opens the developer tools panel, and highlights the "Span" element that holds the text you right-clicked on. Looking at the parent element (the element that contains the spans), you will see the `data-testid="tweetText"` attribute.

Here's an example video showcasing this:



Remember that the "Inspect Element" feature is intended for developers to use for debugging and testing purposes, and changes made using this feature will not be saved when the page is refreshed or closed.

Analyze the Tweets

For the time being, a placeholder can be used inplace for the Sentiment Analyzing function. This placeholder will return a random sentiment value (positive, negative, or neutral). The placeholder will be updated once the sentiment analysis model has been implemented and run. The function below takes in text then returns a sentiment for that text. It also handles adding the sentiment to a **global** object called `tweetSentiment` and sending this updated object to the background script by using the `sendMessage` function.

```
// This function performs sentiment analysis on a given string of text
function analyzeSentiment(text) {
  // For now, just randomly return a sentiment value (1, -1, or 0)
  const sentiments = [1, -1, 0];
  const sentiment = sentiments[Math.floor(Math.random() * sentiments.length)];
  // Save this sentiment value to this text in an object called `tweetSentiment`
  tweetSentiment[text] = sentiment;
  // NOTE: instead of "browser" please use "chrome" if you are planning to run the extension on the chrome browser
  // Send the other scripts the object with all the calculated sentiments
  browser.runtime.sendMessage({
    type: "sentiment",
    data: tweetSentiment,
  });
  // return the calculated sentiment value to use somewhere else
  return sentiment;
}
```

NOTE: The above code is written for FireFox browsers. If you want to deploy to Chrome, replace `browser` with `chrome`.

Getting the Text to Analyze

After collecting the tweets, the text from each tweet should be extracted and sent to the `analyzeSentiment` function. To avoid repeating sentiment analysis for the same tweet, the function can do a check to see if a tweet has already been analyzed. If it hasn't, the sentiment can be calculated and stored as an attribute for the tweet. Otherwise, the stored sentiment value can be used.

```
// This function performs sentiment analysis on the text of a Twitter post
// NOTE: This function does not handle the case with only an image, gif or video
function categorizeTweet(tweet) {
  // check if the tweet's DOM element already has been seen, if it has move on
  if (tweet.hasAttribute("sentiment")) return;
  // Grab all the span elements from the tweet (these contain all the text inside the tweet)
  const spans = tweet.querySelectorAll("span");
  // Create a spans array to hold all the different text in the spans
  const spanTexts = [];
  // Loop through each span and add them to the spanTexts array
  spans.forEach((span) => {
    spanTexts.push(span.innerText);
  });
  // Create a single string from all the elements in the array by joining each string. These are separated by a space.
  const text = spanTexts.join(" ");
  // Check if the text has been analyzed before if it has save the sentiment to the tweet's DOM element so it doesnt need to analyze it again
  if (text in tweetSentiment) {
    const sentiment = tweetSentiment[text];
    tweet.setAttribute("sentiment", sentiment);
    return;
  }
  // If the text hasnt been analyzed, do the sentiment analysis and save the sentiment to the DOM element
  const sentiment = analyzeSentiment(text);
  tweet.setAttribute("sentiment", sentiment);
```

```
}
```

Analyzing all the Tweets on Screen

Currently, the functions above only handle 1 tweet at a time.
These functions need to run for each tweet. Its possible to loop through each tweet, and run the `categorizeTweet` function.

```
// This function categorizes all tweets on the page by performing sentiment analysis on each one
function categorizeAllTweets(tweets) {
  // Loop through each object in tweets and find it's sentiment
  tweets.forEach((tweet) => {
    categorizeTweet(tweet);
  });
}
```

Combining functions

To simplify the process, a single function can be created that combines the steps of grabbing the tweets and running the sentiment analysis on them. This will make it easier to use the extension and avoid repeating code.

```
// This function gets all tweets and then categorizes them all
function doSentimentAnalysis() {
  tweets = document.querySelectorAll('[data-testid="tweetText"]');
  categorizeAllTweets(tweets);
}
```

Analyze on Scroll

To make the sentiment analysis results more dynamic and efficient, the following changes can be implemented:

1. Add a function that analyzes the new tweet that appear on the screen as the user scrolls.
2. Create an object to store the calculated tweet sentiments (prevents recalculating sentiments which saves time).

These enhancements will improve the user experience and make the sentiment analysis more efficient.

```
// Make a global object that stores the current Tweet Sentiments that have been made
tweetSentiment = {};
// Make the Analysis happen everytime the page is scrolled
document.addEventListener("scroll", function () {
  doSentimentAnalysis();
});
```

At this point the `tweet-sentiment.js` file handles:

- retreiving the posts on the screen
- calculating a basic sentiment
- saving that and sending the information to the background script.

Step 3: Sending Information Between Scripts

A background script is used to persist data. While it is not required, it is considered a best practice. This approach allows for efficient calculations and persistence of data, and enables communication from the content script directly to the popup.

The background script is only going to be used for 3 things in this extension.

1. Storing the calculated sentiments for the seen tweets.
2. Calculating the postive, negative, and neutral sentiments from the stored calculated sentiments.
3. Sending the popup the number of positive, negative, and neutral sentiments.

1. Storing the calculated sentiments for the seen tweets

The calculated sentiments for the seen tweets are obtained from the `tweet-sentiment.js` file, specifically from the `sendMessage` call. Specifically, this piece of code from `tweet-sentiment.js`:

```
browser.runtime.sendMessage({
    type: "sentiment",
    data: tweetSentiment,
});
```

It's important to note that this `sendMessage` call includes a type and a data field. The type specifies the script that can add a listener, so in the `background.js` file, the following code can be used to listen for the sent sentiment.

```
let tweetSentiment = {};
// NOTE FOR CHROME: please swap "browser" for "chrome"
browser.runtime.onMessage.addListener(function (message) {
  // Listen for the "sentiment" message, if seen update the `tweetSentiment` object to the data received
  if (message.type === "sentiment") tweetSentiment = message.data;
});
```

2. Calculating the different stored sentiments

As per the goal of displaying the results in an easy to read pie chart, the different sentiments will need to be counted up. So, to calculate the positive, negative, and neutral sentiments, loop through each element in the `tweetSentiment` object and save the sentiment into a `counter` for each value, respectively. To do this efficiently, the `reduce` function can be used.

```
let sentimentValues = [];
function countSentiments(obj) {
  // Use the Object.values method to convert the object into an array of values
  let values = Object.values(obj);
  // Use the reduce method to count the number of occurrences of each value
  let counts = values.reduce(
```

```
    (acc, val) => {
      acc[val == -1 ? String(val) : val]++;
      return acc;
    },
    { "-1": 0, 0: 0, 1: 0 }
  );
  // return the values of the object in an array
  return Object.values(counts);
}
```

3. Sending sentiment data to the popup

The `sendMessage` function will be needed again in the background script to send messages to the popup. To avoid confusion with the content script's `sentiment` message, a different type should be used. The listener from Step 1 can be updated to allow this:

```
// NOTE: instead of "browser" please use "chrome" if you are planning to run the extension on the chrome browser
browser.runtime.onMessage.addListener(function (message) {
  if (message.type === "sentiment") tweetSentiment = message.data; // Set the object to the data received
  // Get the counts of each sentiment value and send it to the other scripts
  sentimentValues = countSentiments(tweetSentiment);
  browser.runtime.sendMessage({
    type: "sentimentValues",
    data: sentimentValues,
  });
});
```

Side note: the listener activates whenever a message is sent to the background. So, whenever a message is sent (from the popup or the content script) the background will always run the `countSentiments` function and send a `sentimentValues` message. This is an implementation detail that can be easily changed to your liking.

Step 4: Displaying the Sentiment Analysis Results

With all the necessary data collected, it can be sent to the popup script and used with Chart.js to display the details in a pie chart. This allows for a quick and intuitive visualization of the sentiment.

Popup Visual

The following code is the `popup.html` code to visualize a simple popup that displays a title and the pie chart.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sentiment Analysis</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.9.3/Chart.min.js"></script>
  </head>
  <body>
    <h1 style="text-align: center; font-size: large">
      Sentiment Analysis Results
    </h1>
    <canvas id="myChart" width="400" height="400"></canvas>
    <script src="./popup.js"></script>
  </body>
</html>
```

How it works:

This line imports Chart.js, allowing the pie chart to be created.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.9.3/Chart.min.js"></script>
```

The body contains an element for the title, the canvas where the pie chart will exist and then the script for the popup is imported and used.

```
<h1 style="text-align: center; font-size: large">
  Sentiment Analysis Results
</h1>
<canvas id="myChart" width="400" height="400"></canvas>
<script src="./popup.js"></script>
```

Popup Javascript

The following code is the `popup.js` code to create the pie chart based off the values received from the background script.

```
// This function create the pie chart using the data provided
function createPieChart(data) {
  document.getElementById("myChart").remove();
  const canvas = document.createElement("canvas");
  canvas.id = "myChart";
  canvas.height = 400;
  canvas.width = 400;
  document.body.appendChild(canvas);
  const ctx = document.getElementById("myChart").getContext("2d");
  const graph = {
    type: "pie",
    data: {
      labels: ["Negative", "Neutral", "Positive"],
      datasets: [
        {
          label: "Sentiment Analysis",
          data: data,
          backgroundColor: [
            "rgba(255, 99, 132, 0.3)",
            "rgba(54, 162, 235, 0.3)",
            "rgba(50, 168, 82, 0.3)",
          ],
          borderColor: [
            "rgba(255, 99, 132, 1)",
```

```
            "rgba(54, 162, 235, 1)",
            "rgba(50, 168, 82, 1)",
          ],
          borderWidth: 1,
        },
      ],
    },
    options: {
      responsive: true,
      maintainAspectRatio: true,
    },
  };
  new Chart(ctx, graph);
  return;
}
sentimentValues = [];
// NOTE: instead of "browser" please use "chrome" if you are planning to run the extension on the chrome browser
// Send a "getSentiment" request to load the sentiments already loaded
browser.runtime.sendMessage({ type: "getSentiment" });
browser.runtime.onMessage.addListener(function (message) {
  // Whenever new sentiments are found, this will trigger updating the sentimentValues data
  if (message.type === "sentimentValues") sentimentValues = message.data; // Set the object to the data received
  // Recreate the pie chart whenever there is any update
  createPieChart(sentimentValues);
});
```

The popup script handles:

- Creating the pie chart in a function
- Asking for the sentiment from the background script
- Retrieves and updates the pie chart

Step 5: Adding a Reset Button

Currently, if you're switching from one topic or search to another all the previously calculated sentiments will still persist. So, adding a reset button to quickly reset the counts of the sentiments will be very useful.

Popup script

In the `popup.js` file, you will need to create a button that tells the background and content script to reset the sentiments that've been calculated.

Add the following code into the `popup.js` file:

```
const reset = document.createElement("button");
reset.style = "display:block; margin: 0 auto; margin-bottom: 0.5rem;";
reset.textContent = "Reset";
reset.addEventListener("click", () => {
  // NOTE: instead of "browser" please use "chrome" if you are planning to run the extension on the chrome browser
  browser.runtime.sendMessage({ type: "reset" });
  browser.tabs.query({ active: true, currentWindow: true }).then((tabs) => {
    // Send a message to the content script in the active tab
    browser.tabs.sendMessage(tabs[0].id, {
      type: "resetSentiment",
    });
  });
});
document.body.appendChild(reset);
```

NOTE: The above code is written for FireFox browsers. If you want to deploy to Chrome, replace `browser` with `chrome`.

The `browser.tabs.query()` function is a way to get information about tabs in a browser extension. When you call the query() function, you can specify certain criteria, such as which tabs you want to get information about. In this lab, it's used to get information from the active tab in the current window.

When the query() function is called, it returns a list of tabs that match the criteria specified. Each tab in the list has some information about it, such as its name, URL, and what window it is in. In this example, `tabs[0]` is the current active tab on the page (a.k.a the current twitter page). This is then used to send a message to the content script to let it know the sentiments need to be reset.

Background script

The `background.js` file has a listener already that is looking for the `sentiment` message type. This function can be improved by adding in a check for a message type of `reset` then setting `tweetSentiment` to an empty object if its been found.

The listener can be updated to look like this:

```
browser.runtime.onMessage.addListener(function (message) {
  if (message.type === "sentiment") tweetSentiment = message.data; // Set the object to the data received
  else if (message.type === "reset") tweetSentiment = {}; // Reset the object to have no sentiments calculated
  // Count the sentiment values and save them into the sentiment values array
  sentimentObject = countSentiments(tweetSentiment);
  sentimentValues = [
    sentimentObject["-1"],
    sentimentObject["0"],
    sentimentObject["1"],
  ];
  browser.runtime.sendMessage({
    type: "sentimentValues",
    data: sentimentValues,
  });
});
```

Content Script

Similarly, the `tweet-sentiment.js` file can be updated to listen for a `resetSentiment` message type, and reset `tweetSentiment` to an empty object if found.

Simply add the following code to the file:

```
browser.runtime.onMessage.addListener(function (message) {
  // reset sentiment if `resetSentiment` is seen
  if (message.type === "resetSentiment") tweetSentiment = {};
});
```

Step 6: Testing your Extension

Testing in FireFox

Step 1: Download the Extension Folder.

There are 2 ways to do this. Either right-click on the `twitter-sentiment-analysis-extension` folder and choose the download option. Or click on the `File` tab and choose the download option.

Remember to unpack/decompress/unzip the downloaded folder. Usually, double clicking on the folder will do the trick.

Step 2: Open a new Firefox browser window.

Step 3: Navigate to this link `about:debugging#/runtime/this-firefox`.

Step 4: Click on `Load Temporary Add-on...` and then find the `manifest.json` file from the downloaded folder of Step 1.

Step 5: Navigate to Twitter.com and log in.

Step 6: Search for a topic or Brand of your interest (e.g. Spotify).

Step 7: Start scrolling and open the popup to view the pie chart load.

Testing in Chrome

Step 0: Change all references of `browser` to `chrome`

To use the extension in Chrome, all instances of `browser` must be changed to `chrome`. These instances are in the Javascript (`.js`) files. So, be sure to double check the `popup.js`, `tweet-sentiment.js`, and `background.js` files.

Step 1: Download the Extension Folder.

There are 2 ways to do this. Either right-click on the `twitter-sentiment-analysis-extension` folder and choose the download option. Or click on the `File` tab and chooose the download option.

Remember to unpack/decompress/unzip the downloaded folder. Usually, double clicking on the folder will do the trick.

Step 2: Open a new Chrome browser window.

Step 3: Navigate to `chrome://extensions/` and ensure the `Developer mode` setting is on.

Step 4: Click on `Load unpacked` and select the entire `twitter-sentiment-analysis-extension` folder.

Step 5: Navigate to Twitter.com and log in.

Step 6: Search for a topic or Brand of your interest (e.g. Spotify).

Step 7: Start scrolling and open the popup to view the pie chart load.

Step 7: Configuring the Natural Language Processing Model

Now that the extension has been verified to work as expected, a pre-trained Natural Language Processing (NLP) model can be connected to perform Sentiment Analysis and provide accurate sentiment scores for the tweets.

To start the model create a new file called `Dockerfile` in the root directory.

```
touch Dockerfile
```

In the `Dockerfile` copy and paste the below code. This code downloads and runs the Sentiment Analysis model.

```
ARG WATSON_RUNTIME_BASE="cp.icr.io/cp/ai/watson-nlp-runtime:1.0.18"
ARG SENTIMENT_MODEL="cp.icr.io/cp/ai/watson-nlp_sentiment_aggregated-cnn-workflow_lang_en_stock:1.0.6"
# Download and unpack the model
FROM ${SENTIMENT_MODEL} as model1
RUN ./unpack_model.sh
# Download the runtime image
FROM ${WATSON_RUNTIME_BASE} as release
# Copy over the model to the runtime image, which enables it to be used
RUN true && mkdir -p /app/models
ENV LOCAL_MODELS_DIR=/app/models
COPY --from=model1 app/models /app/models
```

Now to build the image and run it, you'll need to get an `Entitlement Key` from IBM. Navigate to this page and create a new entitlement key.

Using the entitlement key run the following blocks of code **in the terminal**, remember to replace `key` with your entitlement key that you just copied:

```
IBM_ENTITLEMENT_KEY=key
```

```
echo $IBM_ENTITLEMENT_KEY | docker login -u cp --password-stdin cp.icr.io
docker build -t nlp-model .
docker run -d -e ACCEPT_LICENSE=true -p 8080:8080 nlp-model
```

You can then just click on the "Launch Application" button to start up the Sentiment Analysis model.

Launch Application

Afterwards, you'll see a URL in a browser like setting. Underneath the URL there will be an error message, you can ignore this and continue on. Copy the URL value, so it can be used as the `containerUrl` value in the code below and append `/v1/watson.runtime.nlp.v1/NlpService/SentimentPredict`. An example of the `containerUrl` will be given at the end of this section.

This code can be copied into the `tweet-sentiment.js` file to replace the current `analyzeSentiment` function. Remember to update the `containerUrl` variable.

```
const controller = new AbortController();
const signal = controller.signal;
async function sendRequest(text) {
  const containerUrl = ""; // MUST CHANGE
  // Set the request body
  const postData = {
    rawDocument: {
      text: text,
    },
  };
  // This attempts to send the request to the sentiment analysis model
  // It waits until a response is received
  try {
    const response = await fetch(containerUrl, {
      method: "POST",
      body: JSON.stringify(postData),
      headers: {
        "Content-Type": "application/json",
        "grpc-metadata-mm-model-id":
          "sentiment_aggregated-cnn-workflow_lang_en_stock",
      },
      signal,
    });
    // Parse and save the response from the sentiment analysis model
    const data = await response.json();
    // Handle the response data from the sentiment analysis model
    const label = data["documentSentiment"]["label"];
    if (label == "SENT_POSITIVE") return 1;
    if (label == "SENT_NEGATIVE") return -1;
    return 0;
  } catch (error) {
    // Handle any errors that occurred while making the request, right now just display it in the browser console
    console.error(error);
  }
}
async function analyzeSentiment(text) {
  // send the request with the text that needs to be analyzed
  const sentiment = await sendRequest(text);
  tweetSentiment[text] = sentiment; // save response in an object
  // send the sentiment data out to other scripts
  browser.runtime.sendMessage({
    type: "sentiment",
    data: tweetSentiment,
  });
  return sentiment;
}
```

Here's an example of how `containerUrl` should look:

```
containerUrl = "https://ra-8080.theiadocker-2-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/v1/watson.runtime.nlp.v1/NlpService/SentimentPr
```

You have now connected your browser extension with an actual Sentiment Analysis model. Try testing your browser extension again with this new model connected. Follow the previous steps again to test this updated extension. Ensure that the extension is able to update the pie chart with real values as expected.

Step 8: Publishing your Extension and deploying your NLP model

At this point, the extension and model have been shown to work and communicate together. Let's publish these to make them public!

There are 2 components that are needed to be deployed. An IBM Watson NLP model that will handle your sentiment requests, and the browser extension to their respective addon/extension market.

Deploying the IBM Watson NLP model

Note: Remember to change the `containerUrl` variable's value to the public address thats given to you from the run command.

The `nlp-model` image is setup from Step 7. All thats left is running the image in a public domain. This can be accomplished by deploying the NLP model to IBM Code Engine.

Upload your runtime image to IBM Cloud Container Registry

Step 1: Log in to your IBM Cloud account.

Using the `ibmcloud login` command log into your own IBM Cloud account. Remember to replace `USERNAME` with your IBM Cloud account email and then enter your password when promted to.

```
ibmcloud login -u USERNAME
```

Use `ibmcloud login --sso` command to login, if you have a federated ID.

Then target any specific resource group in your account. By default, if you've completed the sign up process for your IBM Cloud account, you can use the `Default` resource group.

```
ibmcloud target -g Default
```

Step 2: Create a namespace and log in to ICR

You'll need to create a namespace before you can upload your image, and make sure you're targeting the ICR region you want, which right now is `global`.

Choose a name for your namespace, specified as `${NAMESPACE}`, and create the namespace. Currently, it's set to `sentiment-model`, you can choose to rename it to anything you choose.

```
NAMESPACE=sentiment-model

ibmcloud cr region-set global
ibmcloud cr namespace-add ${NAMESPACE}
ibmcloud cr login
```

Step 3: Upload the runtime image to ICR

The following will: set the `REGISTRY` variable to be a global registry called `icr.io`, tag the sentiment analysis model's image from before and then push that image to the registry. You can choose any App name to give your sentiment analysis model running in the cloud. Right now its set to `nlp-model`.

```
APP_NAME=nlp-model

REGISTRY=icr.io
docker tag ${APP_NAME}:latest ${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest
docker push ${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest
```

Deploy the image to Code Engine

IBM Code Engine is a fully managed, serverless platform that runs your continaers on the cloud. Using this tool, the sentiment analysis model can be deployed here and made to use publicly.

However, please note these commands are assuming you have already added a credit card to your IBM Cloud Account and will fail if one has not already been added. Though, you won't be charged until you reach a certain amount of usage. To learn more about IBM Code Engine's free tier, click here.

Step 1: Target a region and a resource group

Choose the region closest to you and/or your target users. Picking a region closer to you or your users makes the browser extension faster. The further the region the longer the request to the model has to travel.

You can choose any region from this list:

Americas

- `us-south` - Dallas
- `br-sao` - Sao Paulo
- `ca-tor` - Toronto
- `us-east` - Washington DC

Europe

- `eu-de` - Frankfurt
- `eu-gb` - London

Asia Pacific

- `au-syd` - Sydney
- `jp-tok` - Tokyo

Use the following commands to target Dallas as the region and the Default resource group.

```
REGION=us-south
RESOURCE_GROUP=Default

ibmcloud target -r ${REGION} -g ${RESOURCE_GROUP}
```

Step 2: Create and Select a new Code Engine project

In this example, a project named `my-sentiment-model` will be create in the resource group set by the previous command.

```
ibmcloud ce project create --name my-sentiment-model
ibmcloud ce project select --name my-sentiment-model
```

Step 3: Create a Code Engine managed secret from the IBM Cloud Web Console

If your applications in a project use an IBM Cloud Container Registry namespace that is in your account, you can let Code Engine create and manage the registry access secret for you to access the namespace from the project. The name of an automatically created registry access secret is of the format: `ce-auto-icr-private-${REGION}`, where `${REGION}` is the region, in which the namespace is created.

Step 4: Create an Code Engine application from the runtime image

```
ibmcloud ce application create \
  --name ${APP_NAME} \
  --port 8080 \
  --min-scale 1 --max-scale 2 \
  --cpu 2 --memory 4G \
  --image private.${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest \
  --registry-secret ce-auto-icr-private-${REGION} \
  --env ACCEPT_LICENSE=true
```

It may take a few minutes to complete the deployment. If the deployment is successful, you'll get the URL of the application's public endpoint from the command output. Append `/swagger` to the URL and open it in a browser to access the Swagger UI, if you want to interact with the REST API resources provided by the Watson NLP Runtime.

Step 5: Check your deployment

You can check the status, logs and events of the application with the following commands.

```
ibmcloud ce app list
ibmcloud ce app logs --application ${APP_NAME}
ibmcloud ce app events --application ${APP_NAME}
```

Done! Remember to change your `containerUrl` variable to the value `${PUBLIC_ENDPOINT}/v1/watson.runtime.nlp.v1/NlpService/SyntaxPredict` where `PUBLIC_ENDPOINT` is set properly to the value thats associated with the Code Engine Deployment.

Finally, test your extension out by following Step 6 again. Essentially, downloading the files and loading them into the browser again.

Deploying the Extension

Step 1: Package the extension into a ZIP file by compressing the source code and any other required assets, such as images and libraries.

Step 2: Open your respective browser and sign into their dashboard.

CHROME: Sign in to the Chrome Developer Dashboard, and create a new project for the extension (register here).

FIREFOX: Sign in to the Firefox Add-ons Developer Hub, and create a new project for the extension (here).

Step 3: Upload the ZIP file to the project, and fill out the required information, such as the name and description of the extension.

Step 4 Publish the extension by submitting it for review and approval.

Once your extension is reviewed and appoved, it will be live and downloadable/used by anyone!

Code Engine Clean up

If you ever wish to stop the Sentiment Analysis model, use the following commands after logging in (these commands assume variables like `NAMESPACE` are still set in the terminal):

Delete the application:

```
ibmcloud ce app delete --name watson-nlp-runtime
```

Delete the project:

```
ibmcloud ce project delete --name my-ce-project --hard
```

Note: If you do not specify the `--hard` option, the project can be restored within 7 days by using either the project restore or the reclamation restore command.

Delete the ICR namespace:

```
ibmcloud cr namespace-rm ${NAMESPACE}
```

Conclusion

This lab has provided you with the tools and knowledge necessary to build a simple browser extension that performs sentiment analysis. By working through the project step-by-step, you have gained practical experience and a deeper understanding of the concepts and techniques involved. Through this hands-on learning experience, you have developed the skills and confidence to continue improving and expanding upon the project on your own. With this foundation, you are now equipped to explore other applications and areas of natural language processing and text analysis.

Thank you for making it to the end of the lab, I hope you had fun creating this extension and learned something new along the way. Happy coding!

Improvements and Next Steps

Creating a Proxy to the Sentiment Analysis Model Deployment

One potential improvement to the extension is to create a proxy to the sentiment analysis model deployment. This would allow the extension to access the model without exposing its API key or other sensitive information. The proxy could be implemented as a separate script that runs on a server, and forwards requests from the extension to the model and vice versa.

Having a UI to Edit the Sentiment Analysis Object

One potential improvement is to provide a user-friendly interface for editing the sentiment analysis object. This could be a form or a menu that allows the user to specify the parameters of the analysis, such as the brand name and the time period to search within. The UI could also provide options for customizing the appearance of the chart, such as selecting the chart type and the colors.

Supporting other Social Media Platforms

One potential improvement to the extension is to add support for other social media platforms, such as Facebook and Instagram. This would allow the extension to analyze posts from these platforms in addition to Twitter, and provide a more comprehensive view of the brand's sentiment. To do this, the extension would need to use the appropriate queries for each platform, and adapt the sentiment analysis algorithms to the specific features and conventions of each platform. For example, the algorithms for Facebook and Instagram would need to account for the use of hashtags and emojis, which are commonly used on these platforms.

Handling different scenarios of text and non-text data

One area where the extension can be improved is in its handling of different scenarios involving text and images. Currently, the extension only processes tweets that contain text in the form of span elements. However, many tweets also contain images, videos, or other non-textual content. In order to make the extension more robust, it would be necessary to add support for analyzing the sentiment of tweets that contain only images, as well as tweets that contain a combination of text and images.

Training the Sentiment Analysis Model

One potential improvement to the extension is to use a high-quality dataset to train and improve the sentiment analysis algorithm. One such dataset is the Sentiment140 dataset from Kaggle, which contains over 1.6 million tweets labeled with their sentiment. This dataset can be used to train a machine learning model that can accurately predict the sentiment of a tweet based on its text. Training the model on Social Media specific posts improves the accuracy and reliability of the sentiment analysis, and provides a more robust and scalable solution for the extension.

Author(s)

Rohit Arora

Changelog

Date	Version	Changed by	Change Description
2022-12-23	1.0	Rohit Arora	Initial version created