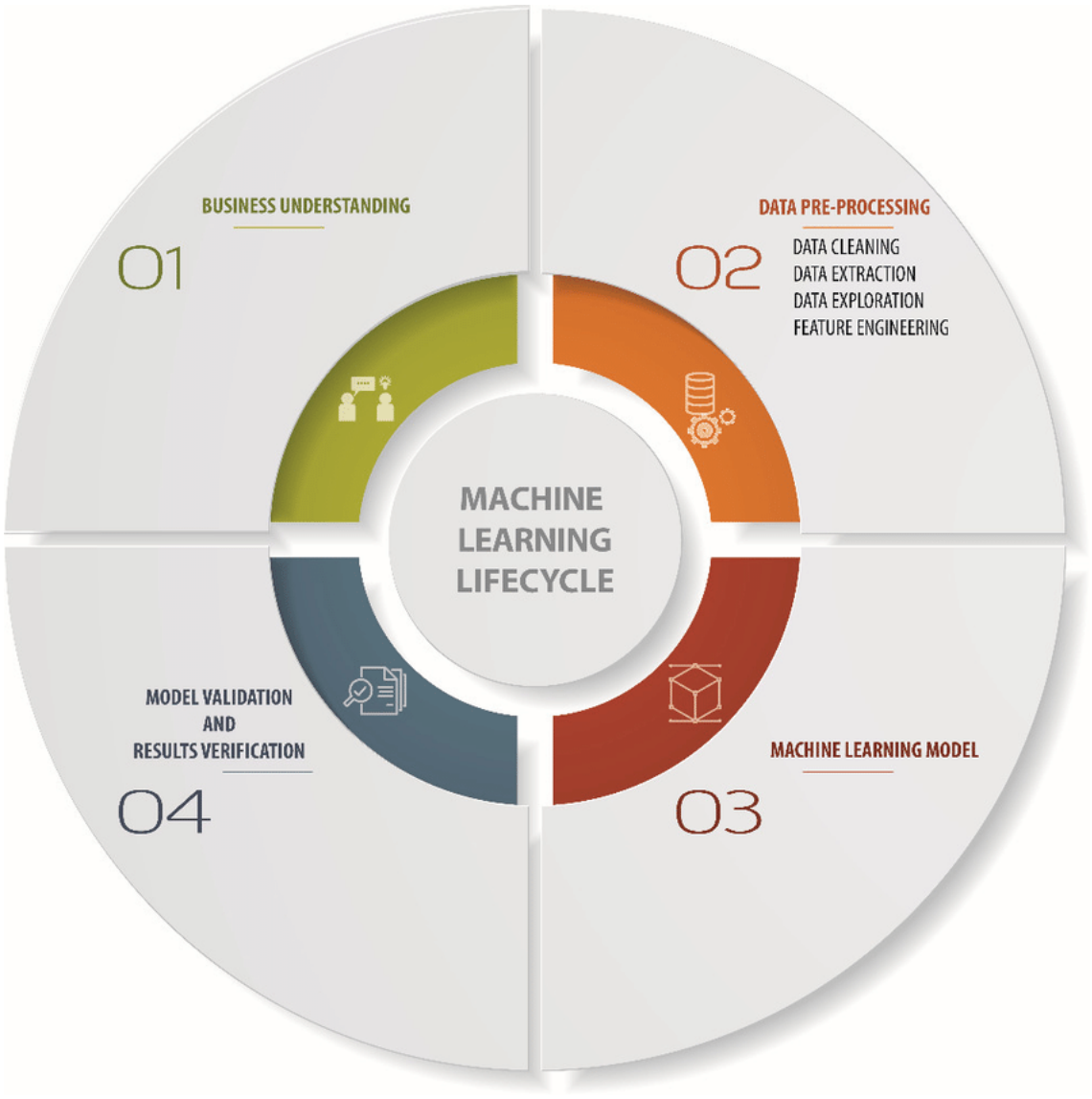
AMAZON-ML HACKATHON 2023

▼ brown.ai.amazon@gmail.com

Brown.ai=amazon

▼ **ML LIFE CYCLE**



▼ **TYPE OF DATA**

▼ **NUMERICAL DATA :** *This type of data includes continuous data (e.g. temperature, height, weight) and discrete data (e.g. count data, such as the number of people in a room).*

▼ **TECHNIQUES**

Cleaning numerical data typically involves identifying and dealing with missing values, outliers, and data that is not in the correct format. Here are some common techniques for cleaning numerical data in Python:

Dealing with missing values: If there are missing values in the numerical data, they can be replaced with a specific value (such as the mean or median), or they can be removed altogether. Here's an example of how to replace missing values with the mean using pandas:

```
pythonCopy code
import pandas as pd

data = pd.read_csv('data.csv')
mean = data['column_name'].mean()
data['column_name'].fillna(mean, inplace=True)
```

Dealing with outliers: Outliers can be identified using statistical methods and removed or replaced with a more appropriate value. For example, you can use the interquartile range (IQR) to identify outliers and replace them with the median value:

```
pythonCopy code
Q1 = data['column_name'].quantile(0.25)
Q3 = data['column_name'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5*IQR
upper_bound = Q3 + 1.5*IQR

data['column_name'] = data['column_name'].apply(lambda x: median if x < lower_bound or x > upper_bound else x)
```

Dealing with incorrect data formats: Sometimes numerical data may be stored as strings or in a different format than what is required. You can convert data to the appropriate format using Python's built-in functions. For example, to convert a string to a float, you can use the float() function:

```
pythonCopy code
data['column_name'] = data['column_name'].apply(lambda x: float(x))
```

Scaling and Normalizing: Numerical data may have different scales and ranges, which can cause issues when training machine learning models. Scaling and normalizing the data can help to improve the performance of the model. Here's an example of how to normalize data using the MinMaxScaler from scikit-learn:

```
pythonCopy code
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
data['column_name'] = scaler.fit_transform(data[['column_name']])
```

Binning: Binning involves grouping numerical data into discrete bins. This can be useful when dealing with continuous data that has a large range. Binning can help to reduce noise and improve the accuracy of predictions. Here's an example of how to bin data using pandas:

```
pythonCopy code
bins = [0, 10, 20, 30, 40, 50]
labels = ['0-10', '10-20', '20-30', '30-40', '40-50']
data['column_name'] = pd.cut(data['column_name'], bins=bins, labels=labels)
```

Feature Scaling: Feature scaling is the process of standardizing the range of features in the data. This can be useful when dealing with features that have different ranges or units. Here's an example of how to perform feature scaling using scikit-learn's StandardScaler:

```
pythonCopy code
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
data[['column1', 'column2']] = scaler.fit_transform(data[['column1', 'column2']])
```

These are just a few examples of how to clean and preprocess numerical data in Python. The specific techniques used will depend on the data and the specific requirements of the analysis.

▼ WHEN TO USE ABOVE TECHNIQUES || ASK QUESTIONS TO CHATGPT

▼ **How to identify when to use the above techniques?**

The choice of techniques to use for cleaning and preprocessing numerical data depends on the specific data set and the problem you are trying to solve. Here are some general guidelines for when to use the techniques mentioned earlier:

1. ***Outlier detection and removal**: If your data set contains extreme values or outliers, it can skew the analysis and lead to inaccurate results. Use outlier detection techniques to identify and remove these values before proceeding with analysis.*
2. ***Missing value imputation**: If your data set has missing values, you may need to impute them before proceeding with analysis. This can be done using techniques like mean imputation, median imputation, or regression imputation, depending on the nature of the data and the missing values.*
3. ***Data transformation**: If the distribution of your data is skewed, it can impact the accuracy of the analysis. You may need to transform the data to a more normal distribution using techniques like log transformation, square root transformation, or Box-Cox transformation.*
4. ***Scaling and Normalizing**: If the numerical data in your data set has different scales or ranges, it can lead to issues when training machine learning models. Scaling and normalizing the data can help to improve the performance of the model.*
5. ***Binning**: If the numerical data in your data set is continuous and has a large range, binning can be used to group the data into discrete bins. This can help to reduce noise and improve the accuracy of predictions.*
6. ***Feature Scaling**: If the numerical data in your data set has different ranges or units, feature scaling can be used to standardize the range of features. This can be particularly important for machine learning models that rely on distance-based algorithms.*

In general, it's important to carefully examine the characteristics of the data and identify any issues that may impact the accuracy of the analysis. Once these issues have been identified, appropriate techniques can be used to clean and preprocess the data before proceeding with analysis.

▼ **how to identify if I need to scale or normalize my data with a graph?**

One way to identify whether you need to scale or normalize your data is by plotting a histogram of your data and examining its distribution. Here's an example of how to create such a plot using the seaborn library:

```
pythonCopy code
import seaborn as sns
import matplotlib.pyplot as plt

# create a sample dataset
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# plot a histogram of the data
sns.histplot(data, kde=False)

# show the plot
```

```
plt.show()
```

This will create a simple histogram plot of the data. You can examine the distribution of the data by looking at the shape of the histogram. If the histogram is roughly symmetric and bell-shaped, then your data may not need to be scaled or normalized. If the histogram is skewed or has a long tail, then you may need to consider scaling or normalizing your data.

For example, let's say that we have a dataset with a skewed distribution:

```
pythonCopy code
import seaborn as sns
import matplotlib.pyplot as plt

# create a sample dataset
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100]

# plot a histogram of the data
sns.histplot(data, kde=False)

# show the plot
plt.show()
```

In this case, we can see that the histogram is skewed to the right, with a long tail extending to the right. This indicates that there are a few extreme values that are much larger than the rest of the data. In such cases, it may be necessary to scale or normalize the data to ensure that these extreme values do not dominate the analysis.

Alternatively, you can also create a boxplot to visualize the distribution of the data and identify any outliers:

```
pythonCopy code
import seaborn as sns
import matplotlib.pyplot as plt

# create a sample dataset
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100]

# plot a boxplot of the data
sns.boxplot(data)

# show the plot
plt.show()
```

This will create a boxplot of the data, which can help you visualize the distribution of the data and identify any outliers. In this example, we can see that there are a few extreme values (50 and 100) that are much larger than the rest of the data. This may indicate that the data needs to be scaled or normalized to ensure that these extreme values do not dominate the analysis.

Regenerate response

▼ **TEXT DATA :** *This type of data includes unstructured text data, such as email messages, social media posts, and news articles.*

▼ **DATA CLEANING**

▼ **Tokenization**: *Tokenization involves breaking the text into smaller pieces or tokens, such as words or phrases. This can help to standardize the format of the text and make it easier to analyze.*

```
import nltk
```

```
from nltk.tokenize import word_tokenize

# Example text
text = "The quick brown fox jumps over the lazy dog."

# Tokenize the text
tokens = word_tokenize(text)

# Print the tokens
print(tokens)
```

```
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', '.']
```

▼ **Stop word removal**: *Stop words are common words that do not provide much meaning to the text, such as "the" or "and." Removing stop words can help to reduce the dimensionality of the text and improve the accuracy of text analysis.*

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Example text
text = "The quick brown fox jumps over the lazy dog."

# Tokenize the text
tokens = word_tokenize(text)

# Remove stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [token for token in tokens if not token.lower() in stop_words]

# Print the filtered tokens
print(filtered_tokens)
```

```
['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog', '.']
```

▼ **Stemming and lemmatization**: *Stemming and lemmatization are techniques used to reduce words to their root or base form. This can help to reduce the variation in word forms and improve the accuracy of text analysis.*

```
import nltk
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize

# Example text
text = "I am running and eating a healthier diet. Starting to get in shape!"

# Tokenize the text
tokens = word_tokenize(text)

# Stem the tokens using Porter stemmer
porter = PorterStemmer()
stemmed_tokens = [porter.stem(token) for token in tokens]
print("Stemmed tokens:", stemmed_tokens)

# Lemmatize the tokens using WordNet lemmatizer
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
print("Lemmatized tokens:", lemmatized_tokens)
```

```
Stemmed tokens: ['I', 'am', 'run', 'and', 'eat', 'a', 'healthier', 'diet', '.', 'start', 'to', 'get', 'in', 'shape', '!']
Lemmatized tokens: ['I', 'am', 'running', 'and', 'eating', 'a', 'healthier', 'diet', '.', 'Starting', 'to', 'get', 'in', '
```

As you can see, stemming reduces words to their base form by removing suffixes, whereas lemmatization reduces words to their base form by using morphological analysis to find the root word. In this example, "run" is

stemmed to "run", whereas it is lemmatized to "running". Similarly, "starting" is lemmatized to "Starting", whereas it is stemmed to "start".

▼ **Spell checking and correction**: *Spell checking and correction methods can help to correct spelling errors in the text. This can be done using tools such as autocorrect or spell checkers.*

```
from spellchecker import SpellChecker

# create a spell checker object
spell = SpellChecker()

# example text with spelling errors
text = "Ths is a sentnce wth spellng errs that ned to be corcted"

# tokenize the text
words = text.split()

# iterate over the words and correct the spelling
corrected_words = []
for word in words:
    corrected_word = spell.correction(word)
    corrected_words.append(corrected_word)

# join the corrected words back into a sentence
corrected_text = " ".join(corrected_words)

# print the corrected text
print(corrected_text)
```

```
" This is a sentence with spelling errors that need to be corrected "
```

In this example, we first create a `SpellChecker` object. We then tokenize the text using the `split()` method and iterate over the words. For each word, we use the `correction()` method of the `SpellChecker` object to correct its spelling. The corrected words are then appended to a list, which is joined back into a sentence using the `join()` method. Finally, the corrected text is printed.

Note that the `SpellChecker` object uses a pre-built dictionary of words to check and correct spelling. If you need to use a custom dictionary or add words to the existing dictionary, you can use the `WordFrequencyDict` class to create a custom dictionary and pass it to the `SpellChecker` object.

▼ **Removing punctuation and special characters**: *Punctuation and special characters, such as commas or hashtags, may not be relevant to the analysis and can be removed to simplify the text.*

```
import re

# example string with punctuation and special characters
string = "This is an example string! It has some punctuation, numbers (like 123), and special characters like @#$%."

# remove punctuation and special characters
clean_string = re.sub(r'[^\w\s]', '', string)

# print the cleaned string
print(clean_string)
```

```
" This is an example string It has some punctuation numbers like 123 and special characters like "
```

In this example, we first import the `re` module for regular expressions. We then define a string with punctuation and special characters. The `re.sub()` method is used to replace all non-word characters (i.e. characters that are not letters, digits, or underscores) with an empty string, effectively removing them from the string. The resulting cleaned string is then printed.

Note that the regular expression `[^\w\s]` matches any character that is not a word character or whitespace character. The `^` symbol at the beginning of the expression negates the match, so the expression matches any

character that is not a word character or whitespace character. The `re.sub()` method then replaces all such characters with an empty string.

- **Removing HTML tags**: *If the text is scraped from a web page, it may contain HTML tags that should be removed.*

```
from bs4 import BeautifulSoup

# example HTML string
html_string = "<html><head><title>Example</title></head><body><p>This is an example paragraph.</p></body></html>"

# create a BeautifulSoup object
soup = BeautifulSoup(html_string, 'html.parser')

# remove HTML tags from the string
text_string = soup.get_text()

# print the cleaned string
print(text_string)
```

```
" ExampleThis is an example paragraph."
```

In this example, we first import the `BeautifulSoup` class from the `bs4` module. We then define an example HTML string. We create a `BeautifulSoup` object with the HTML string and specify the parser to use (in this case, `html.parser`). We then use the `get_text()` method of the `BeautifulSoup` object to extract the text content of the HTML string, which removes all HTML tags. The resulting cleaned string is then printed.

Note that the `get_text()` method preserves the structure of the text (e.g. preserving line breaks and indentation) but removes all HTML tags and their attributes. If you want to remove only certain tags or attributes, you can pass additional arguments to the `BeautifulSoup` object, such as a list of tags to exclude from the parsing.

- **Removing URLs and email addresses :** *URLs and email addresses may not be relevant to the analysis and can be removed.*

```
import re

# example string with URLs and email addresses
string = "This is an example string with a URL https://www.example.com and an email address user@example.com."

# remove URLs
clean_string = re.sub(r'http\S+|www.\S+', '', string)

# remove email addresses
clean_string = re.sub(r'\S+@\S+', '', clean_string)

# print the cleaned string
print(clean_string)
```

```
"This is an example string with a URL  and an email address ."
```

In this example, we first import the `re` module for regular expressions. We then define a string with a URL and an email address. Two regular expressions are used to remove URLs and email addresses respectively. The first regular expression matches any string that starts with `http` or `www` and ends with a non-whitespace character (`\S+`), effectively matching any URL. The second regular expression matches any string that contains the `@` symbol and ends with a non-whitespace character, effectively matching any email address. Both regular expressions replace the matched strings with an empty string, effectively removing them from the original string. The resulting cleaned string is then printed.

Note that the regular expressions used in this example are simple and may not match all possible variations of URLs and email addresses. For more robust solutions, you may need to use more complex regular expressions or external libraries such as `urlextract` for URL extraction and `email-validator` for email address validation.

▼ Cleaning Text Data

Tokenization: Tokenization involves breaking the text into smaller pieces or tokens, such as words or phrases. This can help to standardize the format of the text and make it easier to analyze.

▼ Tokenization

```
import nltk
from nltk.tokenize import word_tokenize

# Example text
text = "The quick brown fox jumps over the lazy dog."

# Tokenize the text
tokens = word_tokenize(text)

# Print the tokens
print(tokens)
```

1. Stop word removal: Stop words are common words that do not provide much meaning to the text, such as "the" or "and." Removing stop words can help to reduce the dimensionality of the text and improve the accuracy of text analysis.
2. Stemming and lemmatization: Stemming and lemmatization are techniques used to reduce words to their root or base form. This can help to reduce the variation in word forms and improve the accuracy of text analysis.
3. Spell checking and correction: Spell checking and correction methods can help to correct spelling errors in the text. This can be done using tools such as autocorrect or spell checkers.
4. Removing punctuation and special characters: Punctuation and special characters, such as commas or hashtags, may not be relevant to the analysis and can be removed to simplify the text.
5. Removing HTML tags: If the text is scraped from a web page, it may contain HTML tags that should be removed.
6. Removing URLs and email addresses: URLs and email addresses may not be relevant to the analysis and can be removed.

▼ **Feature engineering**

Feature engineering for text data involves creating numerical features that can be used as input to machine learning models. This is necessary because most machine learning algorithms operate on numerical data, while text data is typically unstructured and cannot be used directly.

There are several common techniques for feature engineering for text data, including:

1. **Bag of words**: This involves creating a vector for each document or text sample, where the elements of the vector correspond to the frequency of each word in the document. This can be a very high-dimensional vector, so techniques such as term frequency-inverse document frequency (TF-IDF) can be used to weight the words based on their importance.
2. **TF-IDF**: Assign a weight to each word in each document based on its frequency in that document and its rarity across all documents. This approach rewards words that are both frequent and informative, and can be effective for tasks such as topic modeling or text classification.
3. **Word embeddings**: This involves representing words as dense vectors in a lower-dimensional space. This can be done using techniques such as word2vec or GloVe, which learn representations that capture the semantic relationships between words.
4. **Topic modeling**: This involves identifying the underlying topics or themes in a collection of documents. This can be done using techniques such as latent Dirichlet allocation (LDA), which identifies topics as distributions over words.
5. **Named entity recognition**: This involves identifying and extracting entities such as people, organizations, and locations from text data. This can be done using techniques such as named entity recognition (NER).

6. **Part-of-speech tagging**: This involves identifying the part of speech (e.g. noun, verb, adjective) of each word in a sentence. This can be done using techniques such as part-of-speech tagging (POS tagging).

These techniques can be combined and customized depending on the specific problem and dataset. The resulting feature matrix can then be used as input to machine learning algorithms such as logistic regression, random forests, or neural networks.

1. **Categorical data**: This type of data includes nominal data (e.g. color, gender, occupation) and ordinal data (e.g. rating scales, such as "strongly agree" to "strongly disagree").
2. **Time-series data**: This type of data includes data that is collected over time, such as stock prices, weather data, and web traffic data.
3. **Image data**: This type of data includes digital images, such as photographs, drawings, and diagrams.
4. **Audio data**: This type of data includes digital audio signals, such as music, speech, and sound effects.
5. **Video data**: This type of data includes digital video recordings, such as movies, television shows, and surveillance footage.

▼