**Project Documentation for 4 Weeks ML Project Challenge**

**Project Title:**

**Sentiment Analysis on Social Media Data**

**Domain Introduction:**

**Introduction to Machine Learning (ML)**

Machine learning is a subfield of artificial intelligence (AI) that focuses on the development of algorithms and statistical models that enable computers to learn from and make predictions or decisions based on data, without being explicitly programmed for each task. In essence, machine learning algorithms identify patterns and relationships within data, allowing systems to improve their performance over time as they are exposed to more data.

**Key Concepts in Machine Learning:**

1. Data: Data is the foundation of machine learning. It can come in various forms, such as structured data in databases, unstructured data in text documents or images, or semi-structured data in JSON or XML formats. High-quality, relevant data is essential for training accurate machine learning models.

2. Features and Labels: In supervised learning, a machine learning model learns to map input features to output labels based on example data. Features are the input variables used to make predictions, while labels are the outputs or predictions that the model aims to predict.

3. Algorithms: Machine learning algorithms can be categorized into several types, including:

- Supervised Learning: Algorithms learn from labeled data, making predictions or decisions based on input-output pairs.

- Unsupervised Learning: Algorithms learn patterns and structures from unlabeled data, identifying hidden relationships or clusters within the data.

- Semi-supervised Learning: Algorithms learn from a combination of labeled and unlabeled data, leveraging both to improve model performance.

- Reinforcement Learning: Algorithms learn through interaction with an environment, receiving feedback in the form of rewards or penalties based on their actions.

4. Model Training and Evaluation: During the training phase, a machine learning model is exposed to example data to learn the underlying patterns and relationships. The model's performance is evaluated using metrics such as accuracy, precision, recall, F1-score, etc., to assess its effectiveness in making predictions on unseen data.

5. Overfitting and Underfitting: Overfitting occurs when a model learns to capture noise or irrelevant patterns in the training data, leading to poor generalization on unseen data. Underfitting, on the other hand, occurs when a model is too simple to capture the underlying patterns in the data, resulting in low performance on both training and test data.

6. Feature Engineering: Feature engineering involves selecting, transforming, and creating new features from raw data to improve model performance. It plays a crucial role in shaping the input data to make it more suitable for machine learning algorithms.

7. Deployment and Scaling: Once a machine learning model is trained and evaluated, it can be deployed into production environments to make real-time predictions or decisions. Deployment involves considerations such as scalability, performance, security, and monitoring to ensure the model performs reliably in production settings.

**Applications of Machine Learning:**

Machine learning has a wide range of applications across various domains, including but not limited to:

- Natural Language Processing (NLP) for text analysis, sentiment analysis, and language translation.

- Computer Vision for image recognition, object detection, and facial recognition.

- Healthcare for disease diagnosis, personalized treatment planning, and medical image analysis.

- Finance for fraud detection, risk assessment, and algorithmic trading.

- E-commerce for recommendation systems, personalized marketing, and demand forecasting.

- Autonomous vehicles for object detection, path planning, and decision-making.

In summary, machine learning is a powerful technology that enables computers to learn from data and make intelligent decisions, with applications spanning numerous industries and domains. As the volume and complexity of data continue to grow, the importance of machine learning in solving real-world problems and driving innovation will only increase.

**Importance of machine learning projects across various domains:**

1. Enhanced Decision Making:

- Machine learning enables organizations to make data-driven decisions by analyzing large volumes of complex data. Whether it's in finance, healthcare, marketing, or manufacturing, machine learning algorithms can uncover patterns, trends, and insights that humans might overlook, leading to better-informed decisions.

2. Personalization and Customer Experience:

- In industries such as retail, e-commerce, and entertainment, machine learning powers recommendation systems that provide personalized experiences to users. By analyzing user behavior and preferences, these systems can suggest products, content, or services that are highly relevant to individual users, leading to increased engagement and customer satisfaction.

3. Automation and Efficiency:

- Machine learning automates repetitive tasks and processes across various domains, leading to increased efficiency and productivity. For example, in manufacturing, predictive maintenance models can anticipate equipment failures, reducing downtime and maintenance costs. In customer service, chatbots powered by natural language processing (NLP) can handle routine inquiries, freeing up human agents to focus on more complex issues.

4. Healthcare Innovation:

- Machine learning has the potential to revolutionize healthcare by improving diagnostics, treatment planning, and patient outcomes. Algorithms trained on medical imaging data can assist radiologists in detecting anomalies and identifying early signs of diseases such as cancer. Additionally, predictive models can help healthcare providers anticipate patient needs and allocate resources more effectively.

5. Risk Management and Fraud Detection:

- In finance and insurance sectors, machine learning is used for risk assessment, fraud detection, and cybersecurity. By analyzing historical data and identifying patterns indicative of fraudulent activity or security breaches, machine learning models can help organizations mitigate risks and protect against financial losses.

6. Scientific Discovery:

- In fields such as astronomy, biology, and climate science, machine learning is instrumental in analyzing vast amounts of data generated by experiments, simulations, and observations. These algorithms can uncover new insights, patterns, and correlations that lead to breakthroughs in understanding complex phenomena and driving scientific progress.

7. Predictive Maintenance and Asset Optimization:

- In industries like transportation, energy, and utilities, machine learning enables predictive maintenance of critical assets such as vehicles, machinery, and infrastructure. By monitoring equipment performance and predicting failures before they occur, organizations can minimize downtime, reduce maintenance costs, and optimize asset utilization.

8. Social Good and Humanitarian Applications:

- Machine learning projects also have significant applications in addressing societal challenges and promoting social good. From disaster response and humanitarian aid to public health initiatives and environmental conservation, machine learning algorithms can analyze data to provide valuable insights and support decision-making processes aimed at improving the well-being of communities and the planet.

Overall, machine learning projects play a crucial role in driving innovation, efficiency, and progress across diverse domains, ultimately leading to positive impacts on businesses, societies, and individuals.

**Project Lifecycle:**

1. Problem Definition:

- This initial stage involves defining the problem statement and the objectives of the machine learning project. It's crucial to clearly understand the problem domain, stakeholder requirements, and success criteria before proceeding further.

2. Data Collection:

- Once the problem is defined, the next step is to gather relevant data that will be used to train and evaluate machine learning models. Data sources may include databases, APIs, files, or data generated from sensors or devices.

3. Data Preprocessing:

- Raw data often requires preprocessing to clean, transform, and prepare it for analysis. This stage may involve tasks such as handling missing values, encoding categorical variables, scaling numerical features, and removing outliers.

4. Model Development:

- In this stage, machine learning models are selected, trained, and tuned using the preprocessed data. Depending on the problem type (e.g., classification, regression, clustering), various algorithms and techniques may be applied to build predictive or descriptive models.

5. Evaluation:

- Once the models are trained, they need to be evaluated to assess their performance and generalization ability on unseen data. Evaluation metrics such as accuracy, precision, recall, F1-score, or mean squared error are used to measure model performance.

6. Deployment:

- After selecting the best-performing model, it's deployed into production environments to make predictions or decisions in real-time. Deployment involves considerations such as scalability, performance, security, and integration with existing systems.

**Emphasizing Flexibility and Adaptability:**

- The machine learning project lifecycle is iterative in nature, meaning that it often involves multiple cycles of development, evaluation, and refinement. It's rare to achieve optimal results in the first attempt, and it's important to be prepared for experimentation and iteration.

- Flexibility and adaptability are crucial because:

- Data Challenges: Data availability, quality, and relevance can change over time, requiring adjustments to the preprocessing steps or the acquisition of additional data sources.

- Model Selection: Different algorithms may perform better under different circumstances, and it's essential to be open to trying various approaches to find the most suitable one.

- Changing Requirements: Stakeholder requirements or business objectives may evolve during the project lifecycle, necessitating modifications to the problem definition, evaluation criteria, or deployment strategies.

- By maintaining flexibility and adaptability throughout the project lifecycle, teams can respond effectively to challenges, incorporate feedback, and continuously improve the quality and performance of their machine learning solutions.

In summary, the machine learning project lifecycle involves several distinct stages, from problem definition to deployment, with an emphasis on flexibility and adaptability to accommodate changing data, requirements, and circumstances. Iteration and refinement are key to achieving success in machine learning projects.

**Project Development Stages:**

1. Problem Definition:

- The first stage involves clearly defining the problem statement and objectives of the ML project. It's essential to understand the business or domain context, identify stakeholders' requirements, and define success criteria for the project. This stage sets the foundation for the rest of the project lifecycle.

2. Data Collection:

- In this stage, relevant data sources are identified, and data is collected for analysis. Depending on the project requirements, data may come from various sources such as databases, APIs, files, sensors, or web scraping. It's crucial to ensure the quality, integrity, and relevance of the collected data to achieve meaningful results.

3. Data Preprocessing:

- Raw data often requires preprocessing to clean, transform, and prepare it for analysis. This stage may involve tasks such as handling missing values, removing duplicates, scaling numerical features, encoding categorical variables, and feature selection or engineering. Data preprocessing plays a crucial role in improving data quality and making it suitable for ML model training.

4. Model Development:

- In this stage, ML models are selected, trained, and evaluated using the preprocessed data. Depending on the problem type (e.g., classification, regression, clustering), various algorithms and techniques may be applied to build predictive or descriptive models. Hyperparameter tuning and model optimization are also performed to improve model performance.

5. Evaluation:

- Once the models are trained, they need to be evaluated to assess their performance and generalization ability on unseen data. Evaluation metrics such as accuracy, precision, recall, F1-score, or mean squared error are used to measure model performance. Cross-validation techniques may be employed to ensure robustness and reliability of the evaluation results.

6. Deployment:

- After selecting the best-performing model, it's deployed into production environments to make predictions or decisions in real-time. Deployment involves considerations such as scalability, performance, security, and integration with existing systems. Model monitoring and maintenance are essential to ensure continued performance and reliability post-deployment.

7. Iterative Refinement:

- The development of ML projects is often iterative, involving multiple cycles of development, evaluation, and refinement. Feedback from stakeholders, performance metrics, and real-world usage data are used to iteratively improve the models and address any shortcomings. Iterative refinement is key to achieving optimal results and maintaining the relevance of ML solutions over time.

Throughout the project lifecycle, effective communication, collaboration, and documentation are essential to ensure alignment with stakeholder requirements, track progress, and facilitate knowledge sharing within the project team. By following a structured approach to ML project development and addressing each stage systematically, organizations can maximize the chances of success and derive actionable insights from their data.

**Project Title Introduction**

A Twitter sentiment analysis determines negative, positive, or neutral emotions within the text of a tweet using NLP and ML models. Sentiment analysis or opinion mining refers to identifying as well as classifying the sentiments that are expressed in the text source. Tweets are often useful in generating a vast amount of sentiment data upon analysis. These data are useful in understanding the opinion of people on social media for a variety of topics.

**What is Twitter Sentiment Analysis?**

Twitter sentiment analysis analyses the sentiment or emotion of tweets. It uses natural language processing and machine learning algorithms to classify tweets automatically as positive, negative, or neutral based on their content. It can be done for individual tweets or a larger dataset related to a particular topic or event.

**Why is Twitter Sentiment Analysis Important?**

1. **Understanding Customer Feedback:** By analyzing the sentiment of customer feedback, companies can identify areas where they need to improve their products or services.
2. **Reputation Management**: Sentiment analysis can help companies monitor their brand reputation online and quickly respond to negative comments or reviews.
3. **Political Analysis**: Sentiment analysis can help political campaigns understand public opinion and tailor their messaging accordingly.
4. **Crisis Management:**In the event of a crisis, sentiment analysis can help organizations monitor social media and news outlets for negative sentiment and respond appropriately.
5. **Marketing Research:** Sentiment analysis can help marketers understand consumer behaviour and preferences, and develop targeted advertising campaigns.

Twitter sentiment analysis refers to the process of analyzing the sentiment or opinion expressed in tweets posted on the Twitter platform. Tweets are short text messages limited to 280 characters, making them concise expressions of opinions, emotions, or thoughts on various topics. Sentiment analysis on Twitter involves determining whether a tweet expresses a positive, negative, or neutral sentiment towards a particular subject, event, product, etc.

**Here's an overview of the steps involved in Twitter sentiment analysis:**

1. Data Collection: The first step is to collect Twitter data relevant to the analysis. This can be done by accessing the Twitter API or using third-party tools that provide access to Twitter's data stream. Researchers or analysts typically specify keywords, hashtags, or user handles to filter tweets relevant to their analysis.

2. Preprocessing: Once the tweets are collected, they undergo preprocessing steps to clean and prepare the text for analysis. This may include tasks such as removing special characters, URLs, hashtags, mentions, and punctuation, as well as tokenization (splitting text into individual words or tokens) and lowercasing.

3. Feature Extraction: In this step, features are extracted from the pre-processed text to represent the content of the tweets. Commonly used features include word frequencies, n-grams (sequences of adjacent words), and syntactic or semantic features. Feature extraction techniques may vary depending on the specific sentiment analysis approach used.

4. Sentiment Classification: After feature extraction, sentiment classification algorithms are applied to classify the tweets into positive, negative, or neutral categories based on the extracted features. Various machine learning and deep learning techniques can be employed for sentiment classification, including but not limited to:

- Lexicon-based Methods: These methods rely on sentiment lexicons or dictionaries containing lists of words annotated with their sentiment polarity (positive, negative, neutral). The sentiment of a tweet is determined based on the presence and polarity of words in the lexicon.

- Machine Learning Models: Supervised machine learning models, such as Support Vector Machines (SVM), Naive Bayes, Logistic Regression, and Random Forests, are trained on labelled tweet datasets to learn patterns and associations between features and sentiment labels.

- Deep Learning Models: Deep learning architectures, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Transformer models, are capable of learning complex patterns in text data and have shown state-of-the-art performance in sentiment analysis tasks.

5. Evaluation: The performance of the sentiment analysis model is evaluated using appropriate evaluation metrics such as accuracy, precision, recall, F1-score, or area under the Receiver Operating Characteristic (ROC) curve, depending on the task and requirements.

6. Visualization and Interpretation: Finally, the results of the sentiment analysis can be visualized using charts, graphs, or dashboards to provide insights into the overall sentiment trends or patterns observed in the Twitter data. Analysts may also interpret the results to draw conclusions or make informed decisions based on the sentiment analysis findings.

Overall, Twitter sentiment analysis enables organizations, researchers, and analysts to gain valuable insights into public opinion, consumer feedback, market trends, and social dynamics by analyzing the sentiments expressed in tweets on the platform.

**ML Specific Concepts:**

* Dependencies:
  + re
  + pandas
  + numpy
  + seaborn
  + wordcloud
  + train test split
  + confusion matrix
  + classification report
* Logistic Regression
* Bernoulli Naive Bayes
* SVM
* TF-IDF
* Accuracy and F1 score
* NLP using nltk library
* Stemming
* Lemmatization
* Regexp Tokenizer
* ROC AUC Curve

The `re` module in Python provides support for regular expressions, which are a powerful tool for pattern matching and text processing. Regular expressions are a sequence of characters that define a search pattern. They are used to search, match, and manipulate strings based on specific rules or patterns.

Regular expressions can be incredibly useful in machine learning (ML) projects for various tasks such as:

1. Data Preprocessing: Before training a machine learning model, it's often necessary to clean and preprocess the data. Regular expressions can help in tasks such as removing special characters, extracting relevant information, or standardizing the format of textual data.

2. Feature Extraction: Regular expressions can be used to extract features from text data. For example, if you're working with text classification tasks, you might want to extract specific patterns such as email addresses, URLs, or hashtags as features for your model.

3. Text Tokenization: Regular expressions can be used for breaking down text into smaller units called tokens. This is often a crucial step in natural language processing (NLP) tasks where the text needs to be split into individual words or phrases for further analysis.

4. Text Matching and Filtering: Regular expressions allow you to search for specific patterns within text data. This can be useful for tasks such as sentiment analysis, where you might want to identify and categorize words or phrases with positive or negative connotations.

Now, let's discuss some of the key functionalities provided by the `re` module:

1. Pattern Matching: The `re` module provides functions such as `re.match()` and `re.search()` for searching a string for a specified pattern. These functions return a match object if the pattern is found, allowing you to access information about the match.

2. Pattern Compilation: Regular expressions can be compiled into pattern objects using the `re.compile()` function. This allows you to reuse the compiled pattern multiple times, potentially improving performance when working with large datasets.

3. String Replacement: The `re.sub()` function can be used to search for a pattern within a string and replace it with a specified value. This is useful for tasks such as data cleaning and text normalization.

4. Grouping: Regular expressions support grouping of patterns using parentheses. This allows you to extract specific parts of a matched string, which can be useful for tasks such as extracting information from structured text data.

5. Flags: The `re` module supports flags that modify the behavior of regular expression patterns. For example, the `re.IGNORECASE` flag can be used to perform case-insensitive matching.

6. Special Character Escaping: Regular expressions support special characters that have special meanings, such as `.` (matches any character) or `` (matches zero or more occurrences). The `re.escape()` function can be used to escape these special characters if you want to match them literally.

In summary, the `re` module in Python provides powerful tools for working with regular expressions, which can be incredibly useful in various aspects of machine learning projects, including data preprocessing, feature extraction, and text analysis. By mastering regular expressions, you can enhance your ability to work with textual data and improve the performance of your machine learning models.

Pandas is a popular open-source Python library that provides high-performance, easy-to-use data structures and data analysis tools. It is widely used in machine learning (ML) projects for data manipulation, cleaning, exploration, and preprocessing. Here's a detailed explanation of the functionalities provided by the Pandas module for ML projects:

1. Data Structures:

- DataFrame: A two-dimensional labelled data structure with columns of potentially different types. It is similar to a spreadsheet or SQL table, and it is the primary data structure used in Pandas for data manipulation. DataFrames are capable of handling large datasets efficiently and can be created from various data sources such as CSV files, Excel files, SQL databases, or Python dictionaries.

- Series: A one-dimensional labelled array capable of holding any data type (e.g., integers, floats, strings, etc.). Series are the building blocks of DataFrames and represent a single column or row of data.

2. Data Loading and Saving:

- Pandas provides functions to read data from various file formats such as CSV, Excel, JSON, SQL databases, and more. For example, `pd.read\_csv()`, `pd.read\_excel()`, `pd.read\_json()`, `pd.read\_sql()`.

- Data can also be saved to different formats using functions like `to\_csv()`, `to\_excel()`, `to\_json()`, etc.

3. Data Exploration and Manipulation:

- Indexing and Selection: Pandas allows for easy indexing and selection of data subsets using labels or integer-based indices. You can use methods like `loc[]`, `iloc[]`, or boolean indexing to filter rows and columns based on specific conditions.

- Data Cleaning: Pandas provides methods for handling missing data (`dropna()`, `fillna()`), removing duplicate rows (`drop\_duplicates()`), and performing data transformations (`apply()`, `map()`).

- Data Aggregation and Grouping: Pandas supports grouping data based on one or more keys and performing aggregation operations such as sum, mean, count, etc. (`groupby()`).

- Data Visualization: Although Pandas itself doesn't provide visualization capabilities, it seamlessly integrates with other libraries like Matplotlib and Seaborn for data visualization.

4. Feature Engineering:

- Pandas facilitates feature engineering by allowing users to create new features from existing ones. For example, you can create new columns based on mathematical operations, string manipulations, or conditional logic.

- It provides functions for handling datetime data (`to\_datetime()`, `dt.year`, `dt.month`, etc.), which is often used in time-series analysis and feature engineering.

5. Data Transformation:

- Pandas supports various data transformation operations such as merging and joining datasets (`merge()`, `concat()`), reshaping data (`pivot\_table()`, `stack()`, `unstack()`), and sorting data (`sort\_values()`).

- It also offers functions for discretization and binning, scaling and normalization, and encoding categorical variables.

6. Integration with Machine Learning Libraries:

- Pandas integrates seamlessly with popular ML libraries such as scikit-learn, TensorFlow, and PyTorch. DataFrames can be directly used as input for training ML models, and Pandas provides methods to convert data to the required format.

In summary, Pandas is a powerful library for data manipulation and analysis, offering a wide range of functionalities essential for machine learning projects. Its intuitive and user-friendly interface makes it an indispensable tool for data preprocessing, exploration, and feature engineering tasks in ML pipelines.

NumPy (Numerical Python) is a fundamental package for numerical computing in Python. It provides support for multidimensional arrays, mathematical functions for array manipulation, and tools for working with linear algebra, Fourier analysis, and random number generation. NumPy is a critical library in machine learning projects due to its efficiency, flexibility, and wide range of functionalities. Here's a detailed explanation of NumPy's functionalities used in ML projects:

1. Multidimensional Arrays:

- NumPy's primary data structure is the `ndarray`, which represents a multidimensional array of elements of the same type. It is highly efficient for storing and manipulating large datasets commonly encountered in machine learning.

- NumPy arrays support operations such as indexing, slicing, reshaping, and broadcasting, making it easy to manipulate and process data efficiently.

2. Mathematical Operations:

- NumPy provides a wide range of mathematical functions that operate element-wise on arrays, including basic arithmetic operations (addition, subtraction, multiplication, division), exponentiation, trigonometric functions, logarithms, etc.

- These mathematical functions are vectorized, meaning they operate efficiently on entire arrays without the need for explicit looping, resulting in faster computation.

3. Linear Algebra:

- NumPy includes a comprehensive set of linear algebra functions for performing operations such as matrix multiplication, matrix decomposition (e.g., LU, QR, SVD), solving linear equations, computing eigenvalues and eigenvectors, and more.

- Linear algebra operations are crucial in many machine learning algorithms, including regression, classification, dimensionality reduction, and neural networks.

4. Random Number Generation:

- NumPy provides functions for generating random numbers and random arrays with various distributions, including uniform, normal (Gaussian), binomial, and many others.

- Random number generation is essential for tasks such as initializing model parameters, generating synthetic data for simulations, and creating random samples for statistical analysis.

5. Array Manipulation:

- NumPy offers functions for array manipulation, including concatenation (`np.concatenate()`), splitting (`np.split()`), stacking (`np.stack()`), and transposing (`np.transpose()`).

- These functions are useful for rearranging and combining arrays, reshaping data into the desired format, and preparing data for input into machine learning algorithms.

6. Integration with Other Libraries:

- NumPy seamlessly integrates with other Python libraries commonly used in machine learning, such as SciPy, pandas, Matplotlib, and scikit-learn.

- For example, SciPy builds on NumPy's functionality to provide additional scientific computing tools, while scikit-learn leverages NumPy arrays as the primary data structure for implementing machine learning algorithms.

7. Performance Optimization:

- NumPy's array-based computation and vectorized operations are implemented in optimized C and Fortran code, providing high performance compared to pure Python implementations.

- By leveraging NumPy's efficient array operations, machine learning algorithms can process large datasets more quickly and effectively.

In summary, NumPy is a foundational library for numerical computing in Python and plays a crucial role in machine learning projects. Its efficient array-based operations, extensive mathematical functions, and integration with other libraries make it an indispensable tool for data manipulation, mathematical computation, and algorithm implementation in ML pipelines.

Seaborn is a Python data visualization library based on matplotlib that provides a high-level interface for creating attractive and informative statistical graphics. It is specifically designed for visualizing complex datasets and is widely used in machine learning (ML) projects for exploratory data analysis, model evaluation, and result interpretation. Here's a detailed explanation of Seaborn's functionalities used in ML projects:

1. Statistical Visualization:

- Seaborn provides a variety of functions for creating informative statistical visualizations, including scatter plots, line plots, bar plots, histograms, box plots, violin plots, and many others.

- These plots often incorporate statistical summaries such as confidence intervals, means, medians, and quartiles, making it easy to analyze the underlying distribution of the data.

2. Pairwise Relationships:

- Seaborn's `pairplot()` function allows you to quickly visualize pairwise relationships between variables in a dataset. It creates a grid of scatterplots for each pair of variables along with histograms for the individual variables on the diagonal.

- Pairplots are useful for identifying patterns and correlations between features, which is essential in exploratory data analysis and feature selection.

3. Categorical Data Visualization:

- Seaborn provides specialized functions for visualizing categorical data, including `countplot()` for counting the frequency of categorical variables, `barplot()` for showing the relationship between a categorical variable and a numeric variable, and `boxplot()` and `violinplot()` for comparing the distribution of a numeric variable across different categories.

- These plots are helpful for understanding the distribution of data within different categories and identifying any potential outliers or patterns.

4. Regression Analysis:

- Seaborn offers functions for visualizing linear and non-linear relationships between variables using regression plots (`lmplot()`, `regplot()`). These plots display the relationship between two variables along with a regression line or polynomial fit and confidence intervals.

- Regression plots are useful for assessing the strength and direction of the relationship between predictor variables and the target variable in regression tasks.

5. Distribution Visualization:

- Seaborn provides functions for visualizing univariate and bivariate distributions, including histograms (`distplot()`), kernel density estimation plots (`kdeplot()`), and joint distribution plots (`jointplot()`).

- These plots help in understanding the distribution of individual variables and the joint distribution between pairs of variables, which is essential for identifying patterns and outliers in the data.

6. Customization and Styling:

- Seaborn allows for extensive customization and styling of plots, including control over colors, styles, markers, and fonts. It also provides themes for quickly changing the appearance of plots to match different aesthetics or publication requirements.

- Customization options in Seaborn enable users to create visually appealing and publication-quality plots for communicating results and insights from ML projects effectively.

In summary, Seaborn is a powerful data visualization library that complements machine learning workflows by providing intuitive and flexible tools for visualizing complex datasets. Its extensive range of statistical plots and customization options make it an essential tool for exploring, analyzing, and communicating insights from ML projects.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It is widely used in machine learning (ML) projects for data exploration, model evaluation, and result visualization. Here's a detailed explanation of Matplotlib's functionalities used in ML projects:

1. Basic Plots:

- Matplotlib provides functions for creating a wide variety of basic plots, including line plots, scatter plots, bar plots, histograms, pie charts, and more.

- These plots are essential for visualizing data distributions, relationships between variables, and model performance metrics.

2. Customization and Styling:

- Matplotlib offers extensive customization options for controlling the appearance of plots, including colors, line styles, markers, labels, titles, and axis properties.

- Users can customize every aspect of the plot to match their specific requirements or aesthetic preferences, making it suitable for creating publication-quality visualizations.

3. Subplots and Layouts:

- Matplotlib allows for the creation of multiple subplots within a single figure using the `subplot()` function or the `subplots()` function for creating a grid of subplots.

- Subplots are useful for comparing multiple datasets or visualizing different aspects of the data in a single figure.

4. Annotations and Text:

- Matplotlib provides functions for adding annotations, text, and arrows to plots, allowing users to highlight specific features or provide additional context to the visualization.

- Annotations can be used to label data points, add explanatory text, or draw attention to important observations in the plot.

5. 3D Plotting:

- Matplotlib includes support for creating three-dimensional plots using the `mpl\_toolkits.mplot3d` module. Users can create 3D scatter plots, surface plots, wireframe plots, and more.

- 3D plotting is useful for visualizing multidimensional data or exploring complex relationships between variables.

6. Interactive Visualization:

- Matplotlib can be combined with interactive backends such as Jupyter widgets or the `%matplotlib notebook` magic command to create interactive visualizations.

- Interactive visualizations allow users to explore data dynamically, zoom in on specific regions, pan across plots, and interactively adjust parameters, enhancing the data exploration experience.

7. Integration with Other Libraries:

- Matplotlib integrates seamlessly with other Python libraries commonly used in ML projects, such as NumPy, pandas, and scikit-learn.

- Users can easily visualize data stored in NumPy arrays or pandas DataFrames and overlay plots with statistical summaries generated using scikit-learn.

8. Exporting and Saving Plots:

- Matplotlib provides functions for saving plots to various file formats, including PNG, PDF, SVG, EPS, and more.

- Saved plots can be used in reports, presentations, or publications, ensuring that visualizations can be shared and reproduced easily.

In summary, Matplotlib is a versatile and powerful library for creating a wide range of visualizations in Python. Its extensive functionality, customization options, and integration capabilities make it a valuable tool for visualizing data and communicating insights in machine learning projects.

WordCloud is a Python library used for generating word clouds, which are visual representations of text data where the size of each word corresponds to its frequency or importance within the text. Word clouds are commonly used in machine learning (ML) projects for visualizing textual data, identifying key themes or topics, and gaining insights into the most frequent terms in a corpus. Here's a detailed explanation of WordCloud's functionalities used in ML projects:

1. Word Frequency Calculation:

- WordCloud analyzes a given text corpus and calculates the frequency of each word occurring in the text. The frequency of each word determines its size in the resulting word cloud.

- Words that appear more frequently in the text will be displayed with larger font sizes, making them more prominent in the visualization.

2. Customization Options:

- WordCloud offers various customization options to control the appearance of the word cloud, including the shape of the cloud, color palette, font size range, background color, and stopwords (words to be excluded from the word cloud).

- Users can customize these parameters to create visually appealing and informative word clouds that suit their specific needs or preferences.

3. Visualization of Textual Data:

- Word clouds provide an intuitive and visually appealing way to visualize textual data, making it easier to identify prominent words or themes within a corpus.

- ML projects often involve analyzing large volumes of text data, such as customer reviews, social media posts, or survey responses. Word clouds help in summarizing and visualizing this text data quickly and effectively.

4. Text Preprocessing:

- Before generating a word cloud, it is common to preprocess the text data to remove punctuation, stopwords, and perform other text cleaning tasks.

- WordCloud allows users to specify custom preprocessing steps or use built-in functionalities for text cleaning and preparation, ensuring that the resulting word cloud accurately represents the underlying text data.

5. Topic Identification and Analysis:

- Word clouds can be used to identify key topics or themes within a text corpus by visualizing the most frequent words associated with each topic.

- ML projects often involve topic modeling or text classification tasks where understanding the main themes or topics in the text data is essential. Word clouds serve as a useful exploratory tool for gaining insights into the content of the text.

6. Result Interpretation and Communication:

- Word clouds provide a concise and easily interpretable summary of textual data, making them useful for communicating findings and insights to stakeholders or presenting results in reports or presentations.

- ML practitioners often use word clouds as part of their data analysis and visualization pipeline to highlight important trends or patterns in the text data and facilitate decision-making.

In summary, WordCloud is a valuable tool for visualizing textual data in ML projects, offering functionalities for analyzing word frequency, customizing visualization parameters, and gaining insights into the content of the text corpus. Its intuitive interface and versatility make it a popular choice for summarizing and exploring textual data in various domains, including sentiment analysis, topic modeling, and document clustering.

The train-test split is a fundamental technique used in machine learning (ML) projects for evaluating the performance of predictive models. It involves splitting the dataset into two subsets: one for training the model and the other for testing the model's performance. Here's a detailed explanation of train-test split and its functionalities used in ML projects:

1. Dataset Partitioning:

- The train-test split divides the dataset into two disjoint subsets: a training set and a testing set.

- The training set is used to train the model on a portion of the data, while the testing set is kept separate and used to evaluate the model's performance.

2. Evaluation of Model Performance:

- By splitting the dataset into training and testing sets, the train-test split allows for an unbiased evaluation of the model's performance on unseen data.

- The model is trained on the training set and then evaluated on the testing set to assess its generalization ability and ability to make accurate predictions on new, unseen data.

3. Preventing Overfitting:

- The train-test split helps prevent overfitting, a common problem in machine learning where the model performs well on the training data but poorly on new data.

- By evaluating the model on a separate testing set, it provides an indication of how well the model generalizes to unseen data and whether it has learned meaningful patterns or has simply memorized the training data.

4. Parameter Tuning and Model Selection:

- The train-test split is also used for parameter tuning and model selection, where different algorithms or hyperparameters are compared to find the best-performing model.

- Multiple train-test splits or cross-validation techniques are often used for robust evaluation and to ensure the reliability of the results.

5. Cross-Validation:

- Train-test split is a simple form of cross-validation, where the dataset is split into two subsets. However, more advanced techniques like k-fold cross-validation or stratified cross-validation can be used for more reliable performance estimation.

- Cross-validation techniques further improve the robustness of model evaluation by repeatedly partitioning the dataset into training and testing sets and averaging the results across multiple splits.

6. Implementation in ML Libraries:

- Train-test split functionality is provided by ML libraries such as scikit-learn, TensorFlow, and PyTorch.

- In scikit-learn, for example, the `train\_test\_split()` function from the `sklearn.model\_selection` module is commonly used to split the dataset into training and testing sets with specified proportions.

7. Stratified Splitting:

- In classification tasks, it's important to ensure that the class distribution is preserved in both the training and testing sets to avoid biased evaluations.

- Stratified splitting, available in most ML libraries, maintains the proportion of classes in each subset, ensuring that each class is represented in both the training and testing sets proportionally to its occurrence in the original dataset.

In summary, the train-test split is a crucial technique used in ML projects for evaluating model performance, preventing overfitting, and facilitating parameter tuning and model selection. By splitting the dataset into training and testing sets, it enables unbiased evaluation of models and helps ensure their generalization ability to unseen data.

A confusion matrix is a performance measurement tool used in machine learning (ML) projects, particularly in classification tasks. It allows for the visualization of the performance of a classification model by comparing predicted labels to the actual labels in the test dataset. Here's a detailed explanation of confusion matrix and its functionalities used in ML projects:

1. Basic Structure:

- A confusion matrix is typically represented as a square matrix, where each row represents the actual class labels and each column represents the predicted class labels.

- The diagonal elements of the matrix represent the number of data points for which the predicted label matches the actual label, indicating correct predictions.

- Off-diagonal elements represent misclassifications, where the predicted label differs from the actual label.

2. Evaluation Metrics:

- From the confusion matrix, various evaluation metrics can be derived to assess the performance of the classification model, including:

- Accuracy: The proportion of correctly classified data points out of the total number of data points. It is calculated as (TP + TN) / (TP + TN + FP + FN), where TP is true positives, TN is true negatives, FP is false positives, and FN is false negatives.

- Precision: The proportion of true positive predictions out of all positive predictions made by the model. It is calculated as TP / (TP + FP).

- Recall (Sensitivity): The proportion of true positive predictions out of all actual positive instances in the dataset. It is calculated as TP / (TP + FN).

- F1-score: The harmonic mean of precision and recall, providing a balanced measure of model performance. It is calculated as 2 (precision recall) / (precision + recall).

3. Visualization:

- Confusion matrices can be visualized using heatmaps or color-coded matrices, making it easy to interpret the results visually.

- Heatmaps provide a quick overview of the model's performance, highlighting areas of correct and incorrect predictions with different colors or intensity levels.

4. Error Analysis:

- Confusion matrices facilitate error analysis by identifying common types of misclassifications made by the model.

- By examining the confusion matrix, ML practitioners can identify patterns in misclassifications, such as specific classes that are frequently confused with each other, and take corrective actions to improve the model's performance.

5. Model Comparison:

- Confusion matrices enable the comparison of multiple classification models based on their performance metrics.

- ML practitioners can compare the confusion matrices of different models to assess which model performs better overall or in specific classes.

6. Threshold Optimization:

- Confusion matrices can be used to optimize classification thresholds by visualizing the trade-off between precision and recall.

- By adjusting the classification threshold, ML practitioners can tune the model to prioritize precision or recall based on the specific requirements of the application.

In summary, confusion matrices are a valuable tool for evaluating the performance of classification models in ML projects. They provide insights into the model's ability to correctly classify instances and help identify areas for improvement. By analyzing the confusion matrix and derived metrics, ML practitioners can make informed decisions about model selection, parameter tuning, and performance optimization.

A classification report is a summary of the performance of a classification model, typically generated using the predictions made by the model and the actual labels from the test dataset. It provides a comprehensive overview of various evaluation metrics for each class in the dataset, allowing for a detailed analysis of the model's performance. Here's a detailed explanation of a classification report and its functionalities used in ML projects:

1. Metrics Included:

- A classification report typically includes several evaluation metrics for each class in the dataset, including:

- Precision: The proportion of true positive predictions out of all positive predictions made by the model. It measures the model's ability to avoid false positives.

- Recall (Sensitivity): The proportion of true positive predictions out of all actual positive instances in the dataset. It measures the model's ability to detect all positive instances.

- F1-score: The harmonic mean of precision and recall, providing a balanced measure of model performance that takes both false positives and false negatives into account.

- Support: The number of actual occurrences of each class in the test dataset.

- Accuracy: The proportion of correctly classified instances out of the total number of instances in the dataset.

- Macro Average: The average of the metrics calculated for each class, giving equal weight to each class.

- Weighted Average: The average of the metrics calculated for each class, weighted by the number of true instances for each class.

2. Interpretation:

- Each row in the classification report corresponds to a different class in the dataset.

- For each class, the report provides precision, recall, and F1-score, indicating how well the model performs in classifying instances belonging to that class.

- Support indicates the number of instances of each class in the test dataset, providing context for the evaluation metrics.

3. Visualization:

- Classification reports can be visualized as tables or formatted text, making it easy to interpret the results.

- Color-coding or highlighting can be used to draw attention to areas of high or low performance, helping users identify classes where the model performs well or poorly.

4. Model Comparison:

- Classification reports enable the comparison of multiple classification models based on their performance metrics.

- ML practitioners can compare the precision, recall, and F1-score of different models to assess which model performs better overall or in specific classes.

5. Error Analysis:

- By examining the classification report, ML practitioners can identify classes where the model's performance is suboptimal and investigate potential reasons for misclassifications.

- Error analysis can help identify patterns in misclassifications and guide efforts to improve the model's performance through feature engineering, parameter tuning, or model selection.

6. Threshold Optimization:

- Classification reports can be used to optimize classification thresholds by analyzing the trade-off between precision and recall for different threshold values.

- ML practitioners can adjust the classification threshold to prioritize precision or recall based on the specific requirements of the application, as indicated by the classification report.

In summary, a classification report provides a detailed overview of a classification model's performance, enabling ML practitioners to assess its effectiveness, identify areas for improvement, and make informed decisions about model selection, parameter tuning, and performance optimization.

Stemming is a text processing technique used in natural language processing (NLP) to reduce words to their root or base form. It involves removing suffixes from words to obtain their root form, which may not always be a valid word itself but represents the core meaning of the word. Stemming is particularly useful in machine learning (ML) projects for tasks such as text classification, sentiment analysis, and information retrieval. Here's a detailed explanation of stemming and its functionalities used in ML projects:

1. Normalization of Text:

- Stemming helps in normalizing text data by reducing different variations of words to a common base form. For example, words like "running", "ran", and "runner" may all be stemmed to the base form "run".

- Normalizing text data through stemming reduces the vocabulary size and helps in capturing the core meaning of words, which can improve the efficiency and effectiveness of text processing tasks.

2. Feature Extraction:

- Stemming is often used as a preprocessing step in feature extraction from text data. By reducing words to their base form, stemming helps in reducing the dimensionality of the feature space and improving the generalization ability of ML models.

- Stemmed words can be used as features in various ML algorithms, such as decision trees, support vector machines, and neural networks, for tasks such as document classification, topic modeling, and sentiment analysis.

3. Information Retrieval:

- In information retrieval systems, stemming is used to enhance the effectiveness of keyword-based searches by reducing variations of words to a common stem.

- Stemming helps in retrieving relevant documents even if the query terms are slightly different from the terms used in the documents, thereby improving the recall of the retrieval system.

4. Text Preprocessing:

- Stemming is often employed as a part of text preprocessing pipeline along with other techniques such as tokenization, stopword removal, and lowercasing.

- Preprocessing text data using stemming helps in cleaning and standardizing the text data before it is fed into ML models, improving the robustness and performance of the models.

5. Implementation:

- Stemming algorithms such as Porter Stemmer, Snowball Stemmer, and Lancaster Stemmer are commonly used in ML projects for stemming text data.

- Libraries such as NLTK (Natural Language Toolkit), SpaCy, and TextBlob provide implementations of various stemming algorithms in Python, making it easy to integrate stemming into ML pipelines.

6. Limitations:

- While stemming is effective in reducing words to their base form, it may result in the loss of meaning or create non-words in some cases. For example, "happiness" and "happy" may both be stemmed to "happi", which is not a valid English word.

- Stemming algorithms are language-dependent and may not work equally well for all languages or domains. Careful consideration is required when selecting a stemming algorithm based on the characteristics of the text data.

In summary, stemming is a useful text processing technique in ML projects for normalizing text data, reducing vocabulary size, and improving the efficiency and effectiveness of text-based ML tasks. By reducing words to their base form, stemming helps in capturing the core meaning of words and enhancing the performance of ML models in tasks such as text classification, sentiment analysis, and information retrieval.

Both stemming and lemmatization are techniques used in natural language processing (NLP) and machine learning (ML) projects to reduce words to their base or root form. They help in standardizing and normalizing textual data, which can improve the performance of various NLP tasks such as text classification, sentiment analysis, and information retrieval. Here's a detailed explanation of lemmatization and its functionalities:

1. Definition:

- Lemmatization is the process of reducing words to their base or dictionary form, called a lemma. Unlike stemming, which crudely chops off suffixes to derive the root form, lemmatization considers the morphological analysis of words and transforms them to their canonical form.

2. Example:

- For example, the lemma of the words "running", "ran", and "runs" would all be "run". Similarly, "better" would be lemmatized to "good".

3. Part-of-Speech (POS) Tagging:

- Lemmatization often requires knowledge of the part of speech (POS) of each word in the context of the sentence. For example, the lemma of "better" depends on whether it is used as an adjective ("better") or an adverb ("well").

- Some lemmatization algorithms or tools use POS tagging to determine the correct lemma for each word, which can improve accuracy.

4. Normalization:

- Lemmatization helps in normalizing text data by reducing inflected or variant forms of words to a common base form. This can reduce the vocabulary size and improve the performance of NLP tasks by treating different forms of a word as the same entity.

5. Word Sense Disambiguation:

- Lemmatization can help in resolving ambiguities in language by reducing words to their canonical forms, making it easier to determine their intended meaning in context.

- For example, the word "saw" could refer to either a cutting tool or the past tense of "see". Lemmatization can help disambiguate these meanings by transforming "saw" to "see" in the context of a sentence like "I saw a movie."

6. Implementation:

- Lemmatization is implemented using various techniques and libraries in Python, such as the Natural Language Toolkit (NLTK), spaCy, and the TextBlob library.

- These libraries provide pre-trained models and functions to perform lemmatization on text data, often with support for multiple languages and POS tagging.

7. Usage in ML Projects:

- In ML projects, lemmatization is commonly used as a preprocessing step before text analysis tasks such as text classification, sentiment analysis, and information retrieval.

- By reducing words to their base form, lemmatization helps in standardizing the vocabulary and reducing noise in the textual data, which can improve the accuracy and performance of ML models.

In summary, lemmatization is a technique used in NLP and ML projects to reduce words to their base or dictionary form, called a lemma. It helps in normalizing text data, resolving ambiguities, and improving the accuracy of various NLP tasks by treating different forms of a word as the same entity.

A regexp tokenizer, short for regular expression tokenizer, is a tokenizer used in natural language processing (NLP) tasks to split text into individual tokens based on specific patterns defined by regular expressions. Unlike simple whitespace tokenizers or punctuation-based tokenizers, regexp tokenizers provide more flexibility in defining tokenization rules, allowing for finer control over the tokenization process. Here's a detailed explanation of regexp tokenizer and its functionalities used for ML projects:

1. Tokenization:

- Tokenization is the process of splitting text into individual tokens or words. It is a crucial preprocessing step in NLP tasks such as text classification, sentiment analysis, and information retrieval.

- Regexp tokenizers use regular expressions to define patterns for identifying and splitting tokens in text data.

2. Regular Expressions:

- Regular expressions (regex) are sequences of characters that define search patterns. They provide a powerful and flexible way to specify complex patterns for matching text.

- Regexp tokenizers use regular expressions to define patterns for identifying word boundaries, punctuation marks, special characters, or other tokenization rules.

3. Customization:

- One of the main advantages of regexp tokenizers is their flexibility and customizability. Users can define their own regular expressions to match specific tokenization rules or patterns in the text data.

- For example, a regexp tokenizer could be configured to split text on whitespace characters (`\s+`), punctuation marks (`[.,!?;]`), or special symbols (`\W+`).

4. Handling Special Cases:

- Regexp tokenizers can handle special cases or edge cases that may not be handled effectively by simpler tokenization methods.

- For example, they can handle contractions (e.g., "can't" split into "can" and "t"), hyphenated words (e.g., "state-of-the-art" split into "state", "of", "the", and "art"), or numeric expressions (e.g., "USD 100.50" split into "USD" and "100.50").

5. Language-specific Tokenization:

- Regexp tokenizers can be adapted to handle tokenization rules specific to different languages or domains.

- For example, tokenization rules for English text may differ from those for languages with different word boundary conventions or tokenization rules.

6. Preprocessing for ML Models:

- In ML projects, regexp tokenizers are often used as part of the preprocessing pipeline to tokenize text data before feeding it into ML models.

- Tokenization converts raw text into a structured format that ML models can process, making it easier to extract features and perform analysis on text data.

7. Integration with NLP Libraries:

- Regexp tokenizers are commonly available as part of NLP libraries and toolkits such as NLTK (Natural Language Toolkit), spaCy, and scikit-learn.

- These libraries provide pre-built regexp tokenizers along with other tokenization methods and preprocessing tools for use in ML projects.

In summary, regexp tokenizers are a versatile tool for tokenizing text data in ML projects, offering flexibility, customization, and support for handling special cases or language-specific tokenization rules. They play a crucial role in preprocessing textual data before feeding it into ML models, helping to extract meaningful features and improve the accuracy of NLP tasks.

The Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) are evaluation metrics used in machine learning (ML) projects, particularly for binary classification tasks. The ROC curve visualizes the trade-off between the true positive rate (sensitivity) and the false positive rate (1 - specificity) across different threshold values for classification. The AUC represents the area under the ROC curve and provides a single scalar value to quantify the overall performance of a binary classification model. Here's a detailed explanation of the ROC AUC curve and its functionalities used in ML projects:

1. ROC Curve:

- The ROC curve is a graphical representation of the true positive rate (TPR) against the false positive rate (FPR) for a binary classification model across various threshold values.

- The TPR, also known as sensitivity, is the proportion of true positive instances correctly identified by the model, while the FPR is the proportion of false positive instances incorrectly classified as positive by the model.

- The ROC curve plots the TPR on the y-axis and the FPR on the x-axis, with each point on the curve corresponding to a specific threshold value for classification.

2. Interpretation:

- The ROC curve illustrates the performance of the classification model across different threshold values for determining the positive class.

- A model with perfect classification performance would have an ROC curve that passes through the upper-left corner of the plot, indicating a high TPR and a low FPR across all threshold values.

- The closer the ROC curve is to the upper-left corner, the better the model's performance in distinguishing between the positive and negative classes.

3. AUC (Area Under the Curve):

- The AUC represents the area under the ROC curve and provides a single scalar value to quantify the overall performance of a binary classification model.

- The AUC value ranges from 0 to 1, where a higher value indicates better classification performance.

- A model with an AUC of 1 represents perfect classification performance, while a model with an AUC of 0.5 indicates performance comparable to random guessing.

4. Evaluation:

- The ROC curve and AUC are used to evaluate the performance of binary classification models and compare different models.

- ML practitioners use the ROC curve to visualize the trade-off between sensitivity and specificity and assess the model's performance across different operating points.

- The AUC provides a concise summary of the model's discriminative ability, enabling easy comparison of models based on their overall performance.

5. Threshold Selection:

- The ROC curve can help in selecting an appropriate threshold value for classification based on the specific requirements of the application.

- ML practitioners can adjust the threshold to prioritize sensitivity (identifying all positive instances) or specificity (minimizing false positives) based on the application's objectives and constraints.

6. Implementation:

- The ROC curve and AUC are commonly implemented in ML libraries such as scikit-learn in Python.

- ML practitioners can use functions like `roc\_curve()` and `roc\_auc\_score()` to compute the ROC curve and AUC for binary classification models and visualize the results using plotting libraries like Matplotlib.

In summary, the ROC AUC curve is a valuable evaluation metric for assessing the performance of binary classification models in ML projects. It provides insights into the trade-off between sensitivity and specificity and offers a single scalar value (AUC) to quantify the overall performance of the model. By analyzing the ROC curve and AUC, ML practitioners can make informed decisions about model selection, threshold optimization, and performance evaluation in binary classification tasks.

Natural Language Processing (NLP) is a field of artificial intelligence (AI) and computational linguistics focused on enabling computers to understand, interpret, and generate human language in a meaningful way. NLP encompasses a wide range of tasks, including text processing, sentiment analysis, machine translation, speech recognition, and information extraction. Here's a detailed explanation of NLP and the NLTK (Natural Language Toolkit) library and its functionalities used in ML projects:

1. NLP Tasks:

- NLP tasks involve processing and analyzing human language data to extract meaningful information and insights. Some common NLP tasks include:

- Tokenization: Splitting text into individual words or tokens.

- Part-of-speech (POS) tagging: Assigning grammatical tags (e.g., noun, verb, adjective) to words in a sentence.

- Named entity recognition (NER): Identifying and categorizing named entities (e.g., people, organizations, locations) in text.

- Sentiment analysis: Determining the sentiment or opinion expressed in a piece of text (positive, negative, neutral).

- Text classification: Assigning predefined categories or labels to text documents based on their content.

- Language modeling: Predicting the next word in a sequence of words based on context.

- Machine translation: Translating text from one language to another.

- Information extraction: Extracting structured information from unstructured text data (e.g., extracting dates, locations, and events from news articles).

2. NLTK Library:

- NLTK is a leading platform for building Python programs to work with human language data. It provides a wide range of tools and resources for NLP tasks, including corpora, lexical resources, algorithms, and pre-trained models.

- NLTK offers functionalities for performing various NLP tasks, such as tokenization, POS tagging, NER, sentiment analysis, text classification, and more.

- NLTK is widely used in academia and industry for teaching, research, and practical applications in NLP and ML projects.

3. Functionalities of NLTK:

- Tokenization: NLTK provides functions for tokenizing text into words, sentences, or custom patterns.

- POS Tagging: NLTK includes pre-trained models for POS tagging, allowing users to tag words with their grammatical categories.

- Named Entity Recognition: NLTK offers tools for identifying named entities in text, such as people, organizations, and locations.

- Sentiment Analysis: NLTK provides functionalities for analyzing the sentiment of text, including lexicon-based approaches and machine learning models.

- Text Classification: NLTK supports text classification tasks using various algorithms, including Naive Bayes, Maximum Entropy, and Support Vector Machines (SVM).

- Language Modeling: NLTK includes tools for building and evaluating language models, including n-gram models and neural network-based approaches.

- Machine Translation: NLTK provides functionalities for building machine translation systems using statistical and neural machine translation techniques.

- Information Extraction: NLTK offers tools for extracting structured information from unstructured text data, such as regular expressions and dependency parsing.

4. Integration with ML Projects:

- NLTK can be integrated into ML projects as a preprocessing and feature extraction tool for textual data.

- ML practitioners use NLTK functionalities to preprocess text data, extract relevant features, and build models for various NLP tasks in ML pipelines.

- NLTK complements other ML libraries such as scikit-learn, TensorFlow, and PyTorch for building end-to-end NLP and ML solutions.

In summary, NLP is a field of AI focused on processing and analyzing human language data, while NLTK is a popular Python library for working with NLP tasks. NLTK provides a wide range of functionalities and resources for performing various NLP tasks, making it a valuable tool for ML projects involving textual data analysis and processing.

Logistic regression is a statistical method used for binary classification tasks in machine learning (ML) projects. Despite its name, logistic regression is a classification algorithm rather than a regression algorithm. It models the probability that a given input instance belongs to a particular class, typically represented as 0 or 1. Here's a detailed explanation of logistic regression and its functionalities used in ML projects:

1. Model Representation:

- Logistic regression models the relationship between one or more independent variables (features) and a binary dependent variable (target).

- It uses the logistic function (also known as the sigmoid function) to map the input features to a probability value between 0 and 1, representing the likelihood that the instance belongs to the positive class.

3. Decision Boundary:

- The logistic regression model predicts the positive class (1) when the predicted probability is greater than or equal to 0.5 and the negative class (0) otherwise.

- The decision boundary is the hyperplane that separates the instances belonging to different classes in the feature space. It is determined by the values of the model parameters.

4. Training:

- Logistic regression is trained using optimization algorithms such as gradient descent or its variants to minimize a cost function, typically the logistic loss or cross-entropy loss.

- The model parameters (coefficients) are adjusted iteratively during training to minimize the difference between the predicted probabilities and the actual class labels in the training data.

5. Regularization:

- To prevent overfitting and improve generalization, logistic regression models can be regularized using techniques such as L1 regularization (Lasso) or L2 regularization (Ridge).

- Regularization adds penalty terms to the cost function, discouraging large parameter values and promoting simpler models.

6. Evaluation:

- Logistic regression models are evaluated using metrics such as accuracy, precision, recall, F1-score, and ROC AUC score, depending on the specific requirements of the classification task.

7. Implementation Example:

#example demonstrating logistic regression using a static dataset. We'll use the famous Iris dataset, which is available in scikit-learn:

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

# Load the Iris dataset

iris = load\_iris()

X = iris.data[:, :2] # We only take the first two features for simplicity

y = (iris.target != 0) 1 # Convert the target variable to binary: 0 (setosa) vs 1 (versicolor + virginica)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize features by removing the mean and scaling to unit variance

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Fit logistic regression model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Make predictions

y\_pred = model.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

# Print classification report

print("Classification Report:")

print(classification\_report(y\_test, y\_pred))

# Plot decision boundary

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.01), np.arange(y\_min, y\_max, 0.01))

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.RdBu, alpha=0.8)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

plt.title('Logistic Regression Decision Boundary')

plt.show()

In this example:

- We load the Iris dataset and use only the first two features for simplicity.

- We convert the target variable to binary, indicating whether the iris species is not setosa (1) or setosa (0).

- The dataset is split into training and testing sets.

- Features are standardized using `StandardScaler`.

- Logistic regression model is trained on the training data.

- Predictions are made on the testing set.

- We calculate accuracy and print the classification report.

- Finally, we plot the decision boundary of the logistic regression model.

#Sample Example with in-built dataset

import numpy as np

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report

# Generate synthetic binary classification data

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train the logistic regression model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Make predictions on the testing set

y\_pred = model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Classification Report:")

print(report)

In this example:

- Synthetic binary classification data is generated using `make\_classification`.

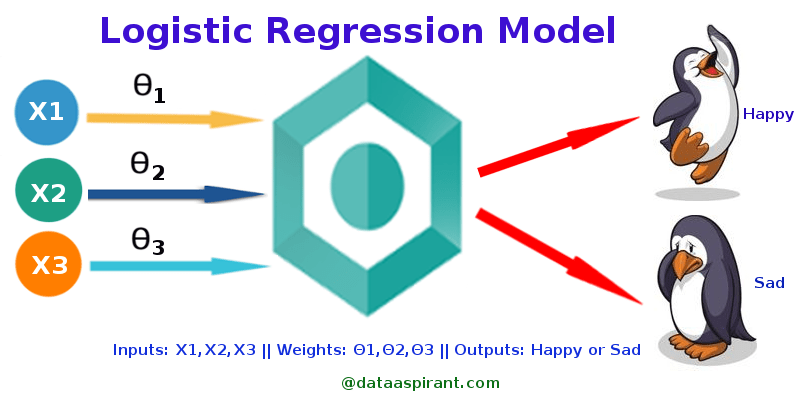
- The data is split into training and testing sets using `train\_test\_split`.

- A logistic regression model is initialized and trained on the training data.

- Predictions are made on the testing set using `predict`.

- The model is evaluated using accuracy and a classification report generated using `classification\_report`.

Theory Part of Logistic Regression:



## What is Logistic Regression?

**Logistic regression** is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. It is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

Logistic Regression is another statistical analysis method borrowed by Machine Learning. It is used when our dependent variable is dichotomous or binary. It just means a variable that has only 2 outputs, for example, A person will survive this accident or not, The student will pass this exam or not. The outcome can either be yes or no (2 outputs). This regression technique is similar to linear regression and can be used to predict the **Probabilities** for classification problems.

## Types of Logistic Regression

#### **Binary logistic regression**

Binary logistic regression is used to predict the probability of a binary outcome, such as yes or no, true or false, or 0 or 1. For example, it could be used to predict whether a customer will churn or not, whether a patient has a disease or not, or whether a loan will be repaid or not**.**

#### **Multinomial logistic regression**

Multinomial logistic regression is used to predict the probability of one of three or more possible outcomes, such as the type of product a customer will buy, the rating a customer will give a product, or the political party a person will vote for.

#### **Ordinal logistic regression**

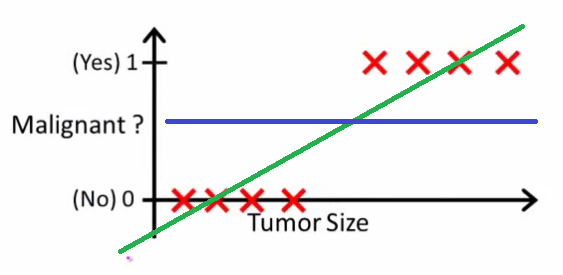
**i**s used to predict the probability of an outcome that falls into a predetermined order, such as the level of customer satisfaction, the severity of a disease, or the stage of cancer.

## Why do we use Logistic Regression rather than Linear Regression?

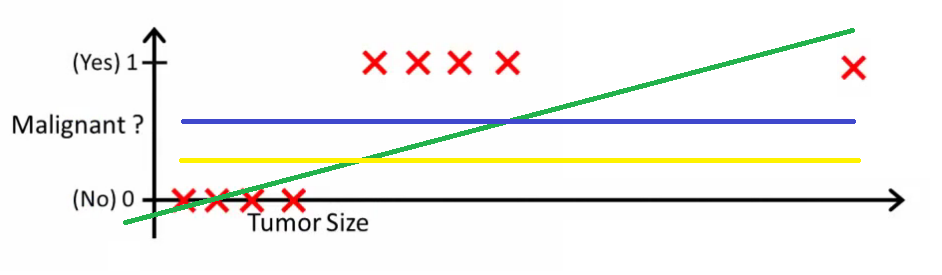
After reading the definition of logistic regression we now know that it is only used when our dependent variable is binary and in linear regression this dependent variable is continuous.

The second problem is that if we add an outlier in our dataset, the best fit line in linear regression shifts to fit that point.

Now, if we use linear regression to find the best fit line which aims at minimizing the distance between the predicted value and actual value, the line will be like this:



Here the threshold value is 0.5, which means if the value of h(x) is greater than 0.5 then we predict malignant tumor (1) and if it is less than 0.5 then we predict benign tumor (0). Everything seems okay here but now let’s change it a bit, we add some outliers in our dataset, now this best fit line will shift to that point. Hence the line will be somewhat like this:



The blue line represents the old threshold and the yellow line represents the new threshold which is maybe 0.2 here. To keep our predictions right we had to lower our threshold value. Hence we can say that linear regression is prone to outliers. Now here if h(x) is greater than 0.2 then only this regression will give correct outputs.

Another problem with linear regression is that the predicted values may be out of range. We know that probability can be between 0 and 1, but if we use linear regression this probability may exceed 1 or go below 0.

To overcome these problems we use Logistic Regression, which converts this straight best fit line in linear regression to an S-curve using the sigmoid function, which will always give values between 0 and 1.

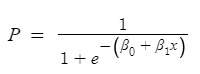
## ****How does Logistic Regression work?****

**Logistic regression works in the following steps:**

1. **Prepare the data: T**he data should be in a format where each row represents a single observation and each column represents a different variable. The target variable (the variable you want to predict) should be binary (yes/no, true/false, 0/1).
2. **Train the model:**We teach the model by showing it the training data. This involves finding the values of the model parameters that minimize the error in the training data.
3. **Evaluate the model:**The model is evaluated on the held-out test data to assess its performance on unseen data.
4. **Use the model to make predictions:** After the model has been trained and assessed, it can be used to forecast outcomes on new data.

### Logistic Function

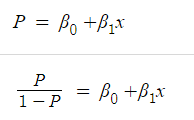
Let’s start by mentioning the formula of logistic function:



 We all know the equation of the best fit line in linear regression is:

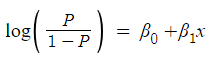
fit linear regression

Let’s say instead of y we are taking probabilities (P). But there is an issue here, the value of (P) will exceed 1 or go below 0 and we know that range of Probability is (0-1). To overcome this issue we take ***“odds”*** of P:

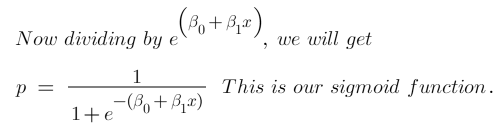
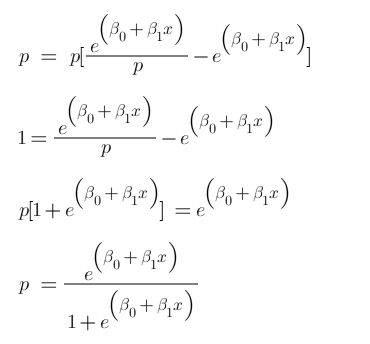
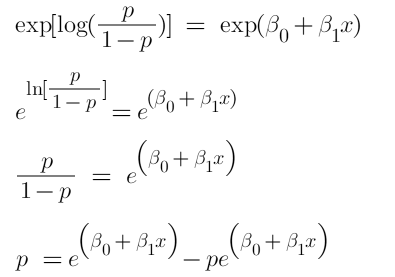


We know that odds can always be positive which means the range will always be (0,+∞ ). Odds are nothing but the ratio of the probability of success and probability of failure. Now the question comes out of so many other options to transform this why did we only take ***‘odds’***? Because odds are probably the easiest way to do this, that’s it.

The problem here is that the range is restricted and we don’t want a restricted range because if we do so then our correlation will decrease. By restricting the range we are actually decreasing the number of data points and of course, if we decrease our data points, our correlation will decrease. It is difficult to model a variable that has a restricted range. To control this we take the ***log of odds***which has a range from (-∞,+∞).



we will multiply by ***exponent*** on both sides and then solve for P.

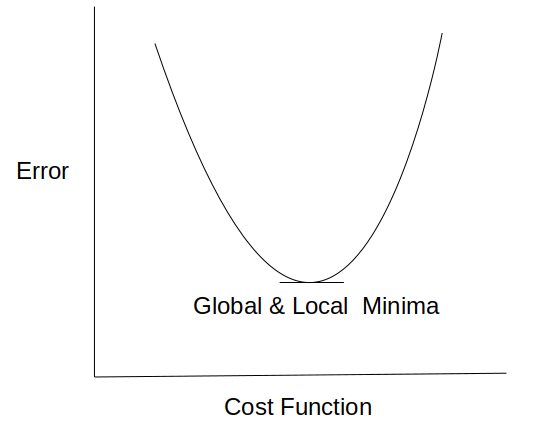


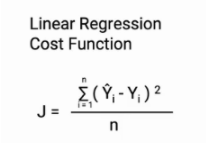
Now we have our logistic function, also called a sigmoid function. The graph of a sigmoid function is as shown below. It squeezes a straight line into an S-curve.



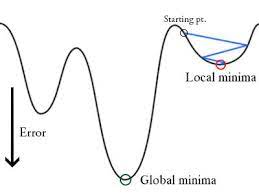
## Cost Function in Logistic Regression

In linear regression, we use the Mean squared error which was the difference between y\_predicted and y\_actual and this is derived from the maximum likelihood estimator. The graph of the cost function in linear regression is like this:



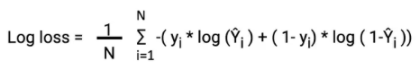


In logistic regression Yi is a non-linear function (*Ŷ*=1​/1+ e-z). If we use this in the above MSE equation then it will give a non-convex graph with many local minima as shown



The problem here is that this cost function will give results with local minima, which is a big problem because then we’ll miss out on our global minima and our error will increase.

In order to solve this problem, we derive a different cost function for logistic regression called ***log loss*** which is also derived from the *maximum likelihood estimation* method.



## What is the use of Maximum Likelihood Estimator?

The primary objective of Maximum Likelihood Estimation (MLE) in machine learning, particularly in the context of logistic regression, is to identify parameter values that maximize the likelihood function. This function represents the joint probability density function (pdf) of our sample observations. In essence, it involves multiplying the conditional probabilities for observing each example given the distribution parameters. In the realm of logistic regression in machine learning, this process aims to discover parameter values such that, when plugged into the model for P(x), it produces a value close to one for individuals with a malignant tumor and close to zero for those with a benign tumor.

Let’s start by defining our likelihood function. We now know that the labels are binary which means they can be either yes/no or pass/fail etc. We can also say we have two outcomes success and failure. This means we can interpret each label as Bernoulli random variable.

A random experiment whose outcomes are of two types, success S and failure F, occurring with probabilities p and q respectively is called a Bernoulli trial. If for this experiment a random variable X is defined such that it takes value 1 when S occurs and 0 if F occurs, then X follows a Bernoulli Distribution.

sigmoid

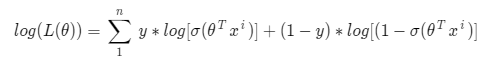
**Where P is our sigmoid function**

sigmoid

where **σ(**θ^Tx^i**)**is the sigmoid function. Now for n observations,

n observations

We need a value for theta which will maximize this likelihood function. To make our calculations easier we multiply the log on both sides. The function we get is also called the log-likelihood function or sum of the log conditional probability

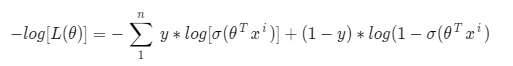


In machine learning, it is conventional to minimize a loss(error) function via gradient descent, rather than maximize an objective function via gradient ascent. If we maximize this above function then we’ll have to deal with gradient ascent to avoid this we take negative of this log so that we use gradient descent.

Also, remember,

***max[log(x)] = min[-log(x)]***

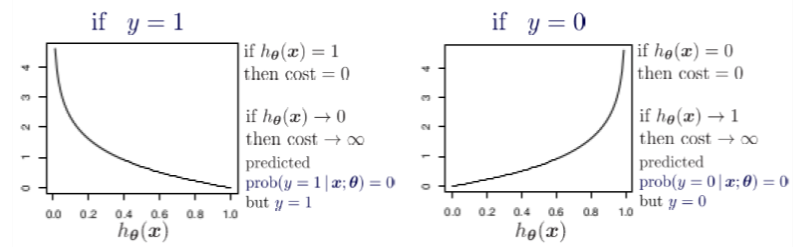
The negative of this function is our ***cost function*** and what do we want with our cost function? That it should have a minimum value. It is common practice to minimize a cost function for optimization problems; therefore, we can invert the function so that we minimize the negative log-likelihood (NLL). So in logistic regression, our cost function is:



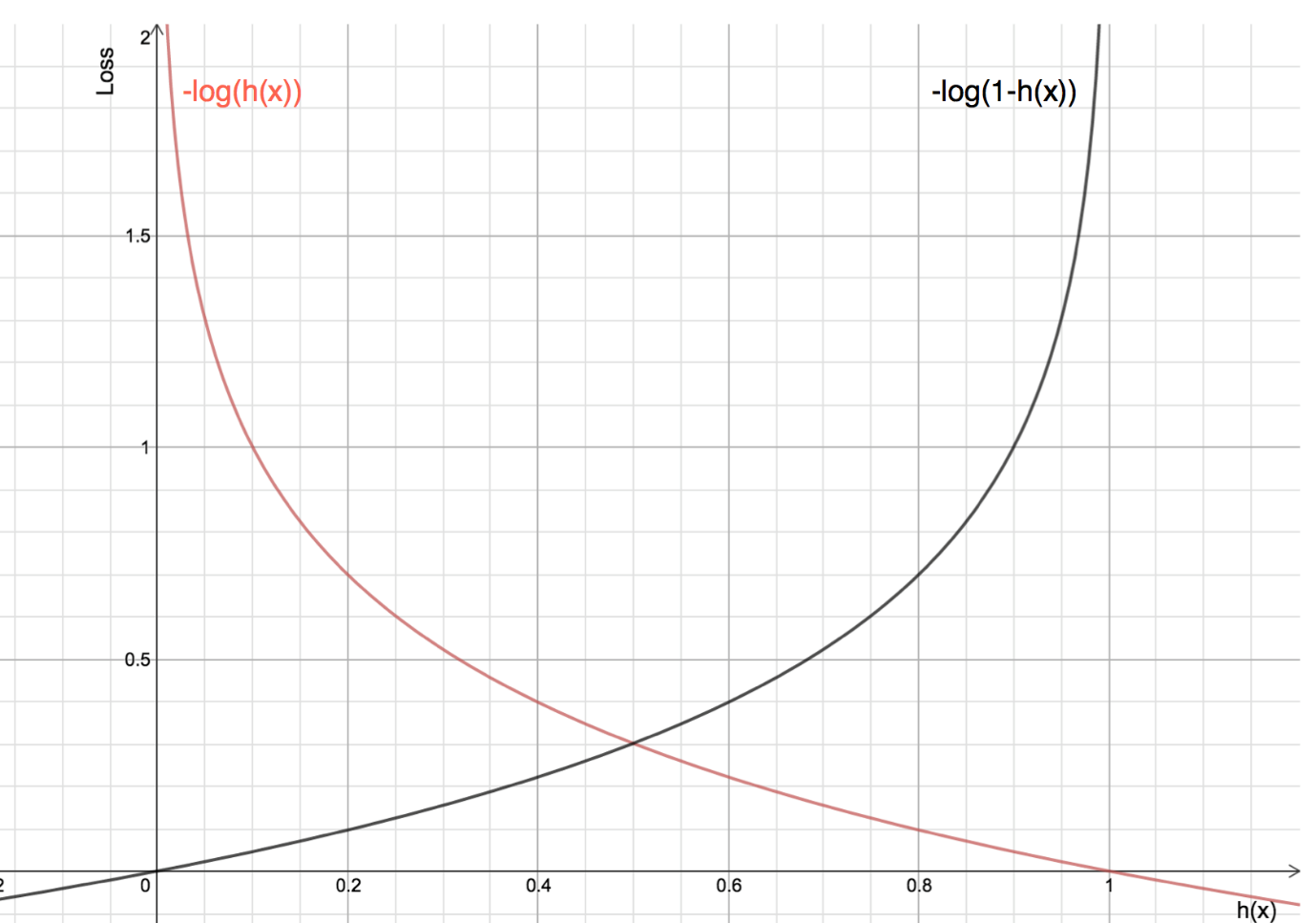
Here y represents the actual class and log(**σ(**θ^Tx^i**) )**is the probability of that class.

* p(y) is the probability of 1.
* 1-p(y) is the probability of 0.

Let’s see what will be the graph of cost function when y=1 and y=0



If we combine both the graphs, we will get a convex graph with only 1 local minimum and now it’ll be easy to use gradient descent here.



The red line here represents the 1 class (y=1), the right term of cost function will vanish. Now if the predicted probability is close to 1 then our loss will be less and when probability approaches 0, our loss function reaches infinity.

The black line represents 0 class (y=0), the left term will vanish in our cost function and if the predicted probability is close to 0 then our loss function will be less but if our probability approaches 1 then our loss function reaches infinity.

cost theta

This cost function is also called log loss. It also ensures that as the probability of the correct answer is maximized, the probability of the incorrect answer is minimized. Lower the value of this cost function higher will be the accuracy.

## Gradient Descent Optimization

Try to understand how we can utilize Gradient Descent to compute the minimum cost.

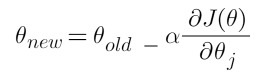
Gradient descent changes the value of our weights in such a way that it always converges to minimum point or we can also say that, it aims at finding the optimal weights which minimize the loss function of our model. It is an iterative method that finds the minimum of a function by figuring out the slope at a random point and then moving in the opposite direction.

![Gradient Descent Optimization, Logistic Regression
](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAeAB4AAD/4SfWRXhpZgAATU0AKgAAAAgABgALAAIAAAAmAAAIYgESAAMAAAABAAEAAAExAAIAAAAmAAAIiAEyAAIAAAAUAAAIrodpAAQAAAABAAAIwuocAAcAAAgMAAAAVgAAEUYc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFdpbmRvd3MgUGhvdG8gRWRpdG9yIDEwLjAuMTAwMTEuMTYzODQAV2luZG93cyBQaG90byBFZGl0b3IgMTAuMC4xMDAxMS4xNjM4NAAyMDIwOjExOjI4IDIwOjI3OjQyAAAGkAMAAgAAABQAABEckAQAAgAAABQAABEwkpEAAgAAAAMwMAAAkpIAAgAAAAMwMAAAoAEAAwAAAAEAAQAA6hwABwAACAwAAAkQAAAAABzqAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMjAyMDoxMToyOCAxNjowOToxMAAyMDIwOjExOjI4IDE2OjA5OjEwAAAAAAYBAwADAAAAAQAGAAABGgAFAAAAAQAAEZQBGwAFAAAAAQAAEZwBKAADAAAAAQACAAACAQAEAAAAAQAAEaQCAgAEAAAAAQAAFioAAAAAAAAAYAAAAAEAAABgAAAAAf/Y/9sAQwAIBgYHBgUIBwcHCQkICgwUDQwLCwwZEhMPFB0aHx4dGhwcICQuJyAiLCMcHCg3KSwwMTQ0NB8nOT04MjwuMzQy/9sAQwEJCQkMCwwYDQ0YMiEcITIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIy/8AAEQgAngEAAwEhAAIRAQMRAf/EAB8AAAEFAQEBAQEBAAAAAAAAAAABAgMEBQYHCAkKC//EALUQAAIBAwMCBAMFBQQEAAABfQECAwAEEQUSITFBBhNRYQcicRQygZGhCCNCscEVUtHwJDNicoIJChYXGBkaJSYnKCkqNDU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6g4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2drh4uPk5ebn6Onq8fLz9PX29/j5+v/EAB8BAAMBAQEBAQEBAQEAAAAAAAABAgMEBQYHCAkKC//EALURAAIBAgQEAwQHBQQEAAECdwABAgMRBAUhMQYSQVEHYXETIjKBCBRCkaGxwQkjM1LwFWJy0QoWJDThJfEXGBkaJicoKSo1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoKDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uLj5OXm5+jp6vLz9PX29/j5+v/aAAwDAQACEQMRAD8A9/ooAKKAK99fWum2M17ezpBbQrvkkc4Cio9L1Wx1rTodQ065juLWYZSRDkGgC5RQAUUAFFABRQAUUAFFABRQAUUAFFABRQAUUAFFABRQAUUAFFAHBfEfwgniKxuL2/v7g6fY2U0i6ejbEeUKSrsRycen/wBeuf8Ahn4Q1PRtM8PazoWo7dO1C1jk1OwuHJQsUH7yP0bI/wA9KAPXaKACigAooAKKACigAooAKKACigAooAKKACigAooAKKACigAooAyPFX/Io6x/15Tf+gGs74b/APJNfDn/AGD4v/QRQB1FFABRQAUUAFFABRQAUUAFFABRQAUUAFFABRQAUUAFFABRQAUUAZHir/kUdY/68pv/AEA1nfDf/kmvhz/sHxf+gigDqKKACigAooAKKACigAooAKKACigAooAKKACigAooAKKACigAooAyPFX/ACKOsf8AXlN/6Aazvhv/AMk18Of9g+L/ANBFAHUUUAFFABRQAUUAFFAGX4hub+00Z5tNjaS6EsICqm8lTIofj/dLfTrXK3XibxaPLMeitHiQFyLeWRfLOzPAG7cAW6elHUBlr4n8U3V7FYvot5AoMLvdG1YAgzIrryu37jMeORtJ96km1jxiniF7COxke1W5MYuPs/yshcSBt3TAiymf7+O9AGbcfEXX7K0jvL/Q3slk/dhZ4JgqvuPJ+Td09MjjNbEet+Ko79IJbDdbnUSjTm3ckW5eUDhR22x/NzwwJ9aAZ29FABRQAUUAFFABRQAUUAFFAGR4q/5FHWP+vKb/ANANZ3w3/wCSa+HP+wfF/wCgigDqKKACigAooAKKACigAooAKKAGvGkgAdFYA5AYZ5p1ABRQAUUAFFABRQAUUAFFAGJL4nsodQvbZ4pxFYsiXFztXy0kYKyp13FiHXopHIGc1F/wmmhNNDDFdmWWSRIwkcbEguSBnjjkHPpQgLPir/kUdY/68pv/AEA1nfDf/kmvhz/sHxf+gigDqKzLbWY7nX73SRE4ktY0kZyeG3en5VUYcyfkJs06KkYUUAFFABRQAUUAFFABRQAUUANdxGjOeijJqppOqW+s6bFf2hbyZCwXeMH5WKnj6g1XK+XmA5nxD4d8QXuutfaXd2yqYtsbTyuDbsFYfKoUg5JUk8Hjv0qC80DxpJPqEcGswNZyJGsCSTEOcY37iIztzzyM/QVKAqW/hbxvYqsNnrNnHbiKZdocjDmPbGf9Wc4bB7Yx36VvaPouu2Wvtd32ptdWht2iCNMeDvJU7doBO04JyMY79gDp6KACigDLu/Dmj31zNcXOnwyTTIEkcjlgOhPuMDB6jA5qOPwroUMgkj0y3WRdmGC8jacjn68+/egBfFX/ACKOsf8AXlN/6Aazvhv/AMk18Of9g+L/ANBFAHUVyll+5+JuqIOfPsIpCfTBIrajtL0Ezq6KxGFFABRQAUUAFFAEN3bi7tJrcyyxCVCnmQuUdc91I6H3rmtP1m60HUIdD8RXIfzTssNSf5Vuf9h+yy/o3bnigDq6KACigCvfv5en3Mn92Jj+hrC8AR+X4I03nO9Xk/76dm/rWy/gv1X6i6li71uZvFVpoenxpK6xm4v5GziCLoo4/jY9B6KTW7WIwooAKKACigAooAyPFX/Io6x/15Tf+gGs74b/APJNfDn/AGD4v/QRQB1Fcjfv9j+JulS5G28s5YOfUfMP5VtQ3a8mTI66isSgooAKKACigAooAKqalpllq9jJZX9ulxbyfeRxn8R6H3oA5uz1S78K3sOj62ZJ9PlfZY6o3IGfuxTHs3YN0bjoevX0AFFAGJ4uuhZ+EtUmJAIt2C59TwP1NVo7+Dwp4Etbi6DMLa1jVY0GXlfACoo7sxwAPet/+XHz/Qn7RL4U0ifTdOkutQCnVb9/tF6wOcOeiA/3VGFH0rfrAoKKACigAooAKKAMjxV/yKOsf9eU3/oBrO+G/wDyTXw5/wBg+L/0EUAdRXIePUa2ttL1lASdOvY3fH9xjtb+lbUP4iFLY61GV0V1OVYZBp1YjKcuq2Fvcvbz3cMUqBWYSMF4bOOvrg0ranYKzK17bBlJUgyrwQMkdeuOaAG/2vpoKg6ha5cAqPOX5ge45pzapp6khr62GH2HMy8N6devtQBVk8SaNHaNdf2jbvEoJPluGOAcHAHJweKWXxFpNvdzWs19FHLCVEm84CliAoz0ycjj3oAsjVNPJUC+tiWfyx++Xlv7vXr7VboAKKAIL2yttRsprO8gSe2mQpJG4yGB7VyEep3XgrUItN1Wc3WizDFneO4MsBH8EueSvQCT6BuuaAOm0TWLbXtGttTtAwhnUkK/VSCVZT7ggj8K0KAOQ8eSNcwaXosWTJqF4gYAf8s0O5j+e39aSNV8T+LgQC+k6E+1Cfuy3g6/URgjn+8T6VtPSlFeoludhRWIwooAKKACigAooAyPFX/Io6x/15Tf+gGs74b/APJNfDn/AGD4v/QRQB1FUtX0+PVdIu7CQZWeIp9D2P51UJcsk+wPYyvBWpNf+HYop2/0uzY206nqGXj9Rg10VXWjy1GhLYwtT8J6dq9+15dGYyNGY8Ajbja65wR1w5/SqL+ANJltpbeWW7kidZFUFkzGH64IXJ55yc9fTishjIvhzosVtHCWuXKEkSMUDH5GXsoHRieB1waZb/DXQ7YXQV7p1uPMyHZDs39cHbnjsSSRQAk/w20i5uUmkur/AAiMiRiRAihmZjgbeuWPPfjOcVoah4OsNT1OS/uJ7nzXVVULsATDK/Hy5blB94nHOMZNAGRD8MNLtpoxFeXgtlcu8Z8sluVIXdszj5Rz973ruaACigArJ17w1o/iazFtq9jHcooOxjkOmeu1hyPwNAHEaxrOkfBHwVDY25uL5pZ5DaQTSDd8x3HJx90Z9O9cp4S/aEfUdcgsdc0yC3guJFjSeFziMk4G4Ht70AdTr+rS3vjZ1sCWvoQdP08DkCdgTLIfaNMk/gK9B0bSrfRNIttOtQfKgTGT1Y9Sx9ySSfrW9fRqPZL/ADJiXqKwKCigAooAKKACigDI8Vf8ijrH/XlN/wCgGs74b/8AJNfDn/YPi/8AQRQB1FFAHFyqfDPjz7SSF07WsI/ok46Z+v8AU12lb1tVGfdfloSuwUVgUFFABRQBQfWLOLWo9Jkdku5YjLEGUhZAOoU9CR1I645q/QAUUAFFAHjnx58E6p4ksNO1TSbeS6lsd6SQRjLFGwdwHU4I6V8923hrVpSZJdPuYbeNgJZpYiip+JHX2q6UeeaiJ6I+ofhZokpsX8R3xZ570H7MX6+Sx3F/q5wc+gFejU6suabYJWQUVmMKKACigAooAKKAMjxV/wAijrH/AF5Tf+gGs74b/wDJNfDn/YPi/wDQRQB1FFAGV4h0SHX9GmsJTtZhmOQdUcdDVLwnrU2oWcljqGE1SxbyrhP72Ojj1B9a3XvUWu36k7M6KisCgooAKKAM3W9DsdfsPsl9GxVWEkciMVeJx0ZWHIIrG0/Wr7RL9NH8SuhEh22OpLwlx/sP/cl/Rs8cjFAHV0UAFY2teJrDQdQ0myu1mabVbn7PAIlBwcZLNkjCjjkZ60AcXf6hdfEXxSdI0m7mt/Delyhr+9hZkNzKpz5SMOw4yf8A6xrL1qZ/iX4+g0K3YnRbA+bdMvRlBwf++jwPYNXRQ91Sqdl+LJfY9fjjSKNY41VEQBVVRgADoAKdXOUFFABRQAUUAFFABRQBkeKv+RR1j/rym/8AQDWd8N/+Sa+HP+wfF/6CKAOoooAK5TxPo1zFdx+I9GU/2lbLiWIHi5i7qR6+n/6q2oSUZ67PRiktDZ0TW7PXtOS8s5MqeHQ8NG3dSOxrSrOcHCTi+gJ3QUVIwooAKrX+n2mqWMtlfQJPbyrteNxwR/Q+4oA5e3vb7wZJHZavNPe6K7iODU5CGe3zwEnPUjOAH9+cV0g1W1Orrpm5vtDwfaF+X5WTOOD3OcfnQBdrx74s6B4i8U6/YHwxJb3EmmQSCeKO7RZYjLx90n5SVHBOOntQBnR+JrnT/Dc/ha38PyaBb6fHi8nSdJck9EVlPzSOSB1zkkHvj0nwH4ZHhvQAs0arfXTedckc4PZM+ijj8/Wuio+Smqfzf6Ere51NFc5QUUAFFABRQAUUAFFAGR4q/wCRR1j/AK8pv/QDWd8N/wDkmvhz/sHxf+gigDqKKACigDktY0O+0zUm13w4qCdubuzbhLgeo9G961tC8RWWvW7NATHcR8TW0nEkR7gj+tdM/wB7TU1utH+jJWjsa9FcxQUUAFFAGV4g0m41nTGtLfUXsmJ5YQpKrjBBV1Ycrz0BHbmuH8OeDfFnh3xdpstxqNnf6LCs0KiKNo3gRhkDaxPyblXABOM8cUAc7D468U6tqWpeDoL62sNRivZhNql26p5MBb5FiTgs2O/b9RvaPp8NklxofgsNJJM2dS1ydt7O3c7v4m5PTgZ/Gt6EE3zS2RMn2NLRdGsbvX1s7JA+k6HLmWRuTcX3qT3KA/8AfR7Yrv6znNzk5MaVkFFQMKKACigAooAKKACigDI8Vf8AIo6x/wBeU3/oBrO+G/8AyTXw5/2D4v8A0EUAdRRQAUUAFc9rPhS31G6Go2U76fqiD5bqAct/vjow+ta0qjpyv0E1dGbH4r1HQpltfFNgyIeF1C1UvE3+8Oq/54rqrO/tNQhE1ncxTxsMho2Bq61HlXPDWLEn0ZZornKCue8aeK4PBnhqbWZ7drhY3RBErbdxY4644oA4fxX4yXxv8O/Esfh7S9Tmt4oF8u7aLy0n+cbtmTuOBz0ziuYg8RL4s02I6/4wIhcCJNC0QMkknbbIzANz+X1q4Qc5JCbsdUPhpb+LXtrjWNGtNMsoUVI4Y1zcyKowPMk69Me/0rqL+O28G+GrPQPDsKx3d0/2awizk7jy0hzyQoyx+gHcVpWcE+WHT8RK+7N/RdJt9D0e2021H7qBcZ7sxJLMfcsST7mr9YFHIeJPGr+H9Wks2sfOQ226Bwxy85J2RnjgEKxz7VVg+JFuYG87T7nzBuy8YUxoQuRklgeTxwPyoWoEifEnT/8Alppmpqoj8wyhIypHzdPnz1Qjp+nNTXHxD02ORlt7C/vFVlUyW4iK5bAXlpB1JA/+tzQBZuPFiLeWaxJEltNafbHeeQq5TP3UUA7m74+nrWXP8UNOS1WWHTL+SV0dljZoVHy4/i3kdDnjPTnFAHbxP5sSSYK7lBwSDj8uKfQAUUAFFAGR4q/5FHWP+vKb/wBANZ3w3/5Jr4c/7B8X/oIoA6iigAooAKKAGvGkilXVWU9QwyDXM3ngTSpbg3WnvcaXdHrJZSFAfqvT8sVrSrSpvTYTVyFNP8aacMQatY6jGOi3cOxsfVf65607+1/GEPEvhy2nxwTBeAZ9wGxxXRbDVNbuLJ95GBrnxPvtBv7fTb3RLe2vrlS0S3OoxRoAP4mYnCj9T2zVS68Xvr2nNZajqHglIZF2ypcajHMpPrt3VKp4eL1kwvIydB8L+D9Llmd/iHYxQTZ32dlqkaQ9c8Asa6y1i+FVpLbTRXvhnz7aUSxTNexNIrjodxbPX3rOpWuuSCshpdzov+E38J/9DRov/gwi/wDiqqt4l8Dvqiam3iHQzeRwmBJTqMeVQkEgDdjkgc+1YFFr/hN/Cf8A0NGi/wDgwi/+Ko/4Tfwn/wBDRov/AIMIv/iqAL9lfaVrURuLC7sr6NWwZIJFlAI7ZGeef1qz9mtwD+5ix3+UUAVdOu9J1a3M+nS2tzCrGMvFhgGUkEcehzVpbS3RdqQRKvoEAoAU28LFC0MZKfcyo+X6elN+xWoAAtoQACANg4B60ATAAAADAHSloAKKACigDO1+zuNQ8O6lZ2uz7RPbSRxeYcLuKkDJ9K870Sz+LGg6HZaTa2fhd4LOFYY2laUsQowM4YDP4UAX/tXxf/58PCf5zf8AxdH2r4v/APPh4T/Ob/4ugA+1fF//AJ8PCf5zf/F0favi/wD8+HhP85v/AIugA+1fF/8A58PCf5zf/F0favi//wA+HhP85v8A4ugA+1fF/wD58PCf5zf/ABdH2r4v/wDPh4T/ADm/+LoAPtXxf/58PCf5zf8AxdH2r4v/APPh4T/Ob/4ugDH17Q/id4hhjW70zwok8TB4bmIyiWJh0KksfyIIOeRVxbT4oKiqdE8FsQMFjFJk+/DUAL9l+KH/AEAvBX/fuT/4qj7L8UP+gF4K/wC/cn/xVAB9l+KH/QC8Ff8AfuT/AOKo+y/FD/oBeCv+/cn/AMVQAfZfih/0AvBX/fuT/wCKo+y/FD/oBeCv+/cn/wAVQBPC3xat12w6X4QiUnOEEoH6PUn2r4v/APPh4T/Ob/4ugDzjSJPGUPxbaPRbfR0uWKnVYtMLm1xuOTJuJAfr05/Wvo2gAooAKKACigAooAKKACigAooARmCqWPQDJ4rEg8YaDcztBFqKGRAjMCjLtDKWUkkYHCk+3egCwfEWkDd/p8Py4x83DZAPynv1HT1pLHxHpOotax219E8t1GZIow3zEAAnjsQCOKANWigAooAKKACigAooAKQgMpB6EYNAGXoHhvSfDFh9i0izS2hLF3xyzse7MeWP1rVoAKKACigAooAKKACigAooAKKACuWf4feHpFlV7WQiVJUfMp5EmM/ljA9BQBnv8MtOivdOmsLp7WO0LbozGr+YDs6dAp+Qc4Nbtl4U0vTr+C9tY5EmhTy1IfgrsVMEd+FX8RQBuUUAFFABRQAUUAFFABRQAUUAFFABRQB//9n/4THkaHR0cDovL25zLmFkb2JlLmNvbS94YXAvMS4wLwA8P3hwYWNrZXQgYmVnaW49J++7vycgaWQ9J1c1TTBNcENlaGlIenJlU3pOVGN6a2M5ZCc/Pg0KPHg6eG1wbWV0YSB4bWxuczp4PSJhZG9iZTpuczptZXRhLyI+PHJkZjpSREYgeG1sbnM6cmRmPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5LzAyLzIyLXJkZi1zeW50YXgtbnMjIj48cmRmOkRlc2NyaXB0aW9uIHJkZjphYm91dD0idXVpZDpmYWY1YmRkNS1iYTNkLTExZGEtYWQzMS1kMzNkNzUxODJmMWIiIHhtbG5zOnhtcD0iaHR0cDovL25zLmFkb2JlLmNvbS94YXAvMS4wLyI+PHhtcDpDcmVhdG9yVG9vbD5XaW5kb3dzIFBob3RvIEVkaXRvciAxMC4wLjEwMDExLjE2Mzg0PC94bXA6Q3JlYXRvclRvb2w+PHhtcDpDcmVhdGVEYXRlPjIwMjAtMTEtMjhUMTY6MDk6MTA8L3htcDpDcmVhdGVEYXRlPjwvcmRmOkRlc2NyaXB0aW9uPjwvcmRmOlJERj48L3g6eG1wbWV0YT4NCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIDw/eHBhY2tldCBlbmQ9J3cnPz7/2wBDAAMCAgMCAgMDAwMEAwMEBQgFBQQEBQoHBwYIDAoMDAsKCwsNDhIQDQ4RDgsLEBYQERMUFRUVDA8XGBYUGBIUFRT/2wBDAQMEBAUEBQkFBQkUDQsNFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBT/wAARCAF+AmoDASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwD9U6KKKACiiigAooooAKKKKACiiigAooprNt9KAHUVGZCvUU9ST1GKAFooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooApaxrFl4f0q81PU7y307TbKF7m6vLuVYoYIkUs8juxAVVUEliQAASa/IP/AIKFf8FMo/i1pc/w4+EGo3EXg28gC614iEUttNqasuTaxK6q8cGDiRmAaQ5TAjDed9a/8FP/AII/Gn48fDnwt4f+FQ/tHRvtkr+IdDjvobN7zARrWRmlKK8UbJKSm/77xNsYoGj+f/jd+wR4Q/ZU/wCCdfj/AFDUbXT/ABL8TZf7MnvPEcluH+xudQtkMFiXXdFGFkdS42vLuYvhdkcYB4X/AME9/wDgoTffs06rB4H8b3FxqPwrvZiVcBpJtBldstNEo+ZoGYlpIVGQSZIxu3rL+1/hbxdonjbQbbW/Dmsaf4g0a6LfZ9Q0u6S5t5trMjbJEJVsMrKcHgqR1FfkH+xH+w34Q/bB/Y91try6/wCEY8Z6b4zuY7HxNb2oncQG0sS9vNFuTzojlmUbgY3O5SA0iycX8Efjb8T/APgl78eNT8C+O9Nn1DwfdTLNqekwSF4bqJjsTU9Od9qliqY52iQIYpNjxqYgD9yFO4Zpaav3adQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAm0ZzjmvlX/gqMMfsJ/E3/uGf+nO0r6rr5V/4Kj/8mJ/E3/uGf+nS0oA8p/4IrKG/ZZ8Vf9jndf8ApDY19veI/h34V8Yapoup6/4a0fXNS0Sb7Tpd5qVhFcTWEu5G8yB3UmJt0cZyhByinsK+Iv8Agir/AMms+Kv+xzuv/SGxr7/oARVCjApaKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACvlX/AIKj/wDJifxN/wC4Z/6dLSvqqvlX/gqP/wAmJ/E3/uGf+nS0oA8q/wCCKv8Ayaz4q/7HO6/9IbGvv+vgD/gir/yaz4q/7HO6/wDSGxr7/oAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACvlX/gqP/wAmJ/E3/uGf+nS0r6qr5V/4Kj/8mJ/E3/uGf+nS0oA8q/4Iq/8AJrPir/sc7r/0hsa+/wCvgD/gir/yaz4q/wCxzuv/AEhsa+/6ACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAr5V/4Kj/8mJ/E3/uGf+nS0r6qr5V/4Kj/APJifxN/7hn/AKdLSgDyr/gir/yaz4q/7HO6/wDSGxr7/r4A/wCCKv8Ayaz4q/7HO6/9IbGvv+gAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooA8d8XftCf8It+0t4K+Ev9hC5/wCEk0q61P8Atj7Zt+zeSGOzyfLO/dt67xjPSuntvjt8N77UNKsLb4g+Fbi/1ZmTTrWLWrZpbxldo2EKiTMhDo6naDhlI6givD/ih4P8QX3/AAUE+EfiK10TUrjw/ZeGdTgutWitJGtLeRxJsR5Qu1WbjCk5Oa+TX/ZPkj/YV1bU0+E99/wtl/F32uOX+w5W1kQjUgAUGzzRF5OTgDaR83vURff+tQlvp/Wh9qePP2xtJ+FGqfFB/GOm2FhoHgxLIwXOn+IbG61DU5LiMv5QsS6SQOMHaJSPMALLwK09M/aq0nxh4i+F0Hg9ND8RaD40W4M9+fE9lDdaa8UCS+ULTez3Mo3hZEiJMXVuDXyt8XP2f/FXxE8cftiongi91I6tpeizeHri809vKvLmC2O82krrtaRfnXKHILbf4sU7wP4R1TVfH37JuqeGPg/4k8E2Ph6PWLfWbfUfDh05UvPsEMZuZxGCqLNIpCyS7WcjpVLzHa70/rY+59H+NXgDxF4pk8MaT468M6n4ljLrJo1nq9vNeIU++DCrlwV78cd686/Zb+PfiD43XXxUj12z021Xwr4xvvD1l/Z0UieZbw7djS73bLnccldo9AK/Or4c/Cf4p6j8ePhjq+pfCzWPDEuk+NLWTVLPSfhpaaTpunxecw3RanCzXF5CFTlpRsGclz8pb7k/YZ8Ha/4R1H47Pruh6loqah8RNUvrJtQtJLcXVu/l7Jo96jfG2DhhkHHWqUdLt9CZO2i7ouaR+29bzfCX40eL9Y8Kf2PqXwz1S80u60U6gZPtbxBfJYS+SuwTM20fI23Geaq6b+3Zba9+zv8ADf4k6T4RN5qnjbX7fw5beHjqWPIupJpI23TCEkqoiZ/9WCQRwM5rwr9oL4JeOpf2tNb8L6H4V1jUfh18Vb7w/qOvavbWsslnZCxmYzrJIq7Y2dY1OGYZ38dhU3wq+BvjTTf2yz4MvvC2qQfDHwn4p1bxzpmtSWTjT5Xu7eAW9tDKVEZMbySNtXkFW/uk1EPeSb6/puOWm39dj7r1P4t+CtD8WWfhbU/GGgad4nvNv2fRLrU4Ir2bdkLshZ97ZwcYHOK5jw78bm1D4mfEbw3rFlo2jaN4Qhtbj+2f+EjtZ5JUliMjtcWqkSWapjhpsBxlhwK/LT41fs3/ABBh+K3xRsPE+geO7yLXPElxqlnrHhX4X2niUy27yB4WTU2niuLcqAqmGNgBtOfvEV6b+0t4R8QeGLj9qq81nR9YOn69/wAIfZadqc1kUXU5o2h8z7Or/JdOGUloVY5IIJGMh30Ui+XW1z70139prwHqHg3xbeeBvHnhDxZ4g0fRbzVYtP07WILxv3MRYM8cUm7Zu2gkY69RkVR/Zx/aU0T4vfD/AMEy6x4i8N23j/W9Hi1O48O2N6iXC7l3MyWzSNKE6nnPHevkD4Dt4d8T698cvt+h+Kbb41614Jn8nS9a8CR+HWNgkBhH2W3ilnbc8jRhiXy5ChVwvCaD+y6vhD4ZfskaxofwwudM8Z2vifTrjxLe2ujSrfwxMjvM1423eiBwufMwq8AYHFUo+/yvyIbXLddz7B+E/wC198N/jJ8RvFXgnw/rUH9uaDdfZRHPd22NSwrM8tmEmZpo12nc20Y4zXtStmvhj9mv4Xw/CP8AbO+LVlf/AAgvrSHWtQ/tHw54us9Bh/suxtPJbzIluxjyWcuF8pBknO4CvuVP885pdEyb+810H0tIDS0FBRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAV8q/8ABUf/AJMT+Jv/AHDP/TpaV9VV8q/8FR/+TE/ib/3DP/TpaUAeVf8ABFX/AJNZ8Vf9jndf+kNjX3/XwB/wRV/5NZ8Vf9jndf8ApDY19/0AFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFADWXcc0bB9adRQA3yxR5YyTzTqKAG7BR5Y645p1FADTGDRsHvTqKAG7BXKfE74UeE/jN4Pu/C3jTRbfXtBuirSWk5ZfmU5VldSGRh/eUg8nnk11tFJq+4LTY8n+Cn7Kvwr/Z1kv5fh74Rt/D9zfgLc3XnzXM7qMHZ5szu4TIB2ggZGcZr1bYKdRVCGeWOPUd6dtFLRSGJS0UUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFACE0m6lNMJwwo8gHilqNZOv1p6nOaAClpjHnHel4xQA6vlX/gqP/yYn8Tf+4Z/6dLSvqmvlb/gqP8A8mJ/E3/uGf8Ap0tKAPKv+CKv/JrPir/sc7r/ANIbGvv+vgD/AIIq/wDJrPir/sc7r/0hsa+/6ACiiigBDmk3Glao2kCdTgUgJKWmodw65p1MAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKazbe1AC5paZu74zS5/Cl1sAtcr8SPil4U+EHhe48R+M9fsfDuiwHa91eybQzEEhEH3ncgHCKCxxwK6g5r4n/bgudJ8M/tHfs7+KfH9uJfhZp99qEOoXF1AZbK0vpIVFrJcDBXG8ZUkDGxjnANS3qlfcaW57x8Ff2vPhL+0Pqt7pfw/8ZW+vanZw/aJrM2txayrHuClws8aFlBZQSoIG4Z6ivXvMPHFfnWf2q/GvjTxx8Y/DN5qnwy1n+xvhprWtaN4u+GtxPPdWuCFihN2ZTscfK7KgGGEZBrD8Sa9rfwy/Y18GeNPEPxY+KOv+JfiO2kov2bxPa6XFbP8AZ5XWKO7njZbGMqQZZjuZ2iUsQCxq3dK4JXdj9Mt5x059K868YfHbQfBPxa8GfD2/tdQfWPFVve3VncQRI1tElrGJJPNYuGBII27VbJ64r83vhf8AtL/FTQ/2X/2kprLxRrOpX3hefTU0i81HX4PEt1p8dzJsuHGoQrsnAUFgyjanbBBz03w/Xw7YftmfBC38L/HPWPjTEuhaxdTXGsasmpTafNJZHIWZRuQPtz5LktHs/wBo07a2J3Tsfol8K/i14X+NXguz8W+DdT/tnw/dvJHDefZ5YN7RuUcbJVVhhlI5HOK6wS9D1HrX49ah+0V8Vl/Zh+A9n/wlniKWz8S6jrg1jXV8WRaNfXDQXDeRD/a92HW3AGT833goQdq7ix+Kvxik/ZR0xbrx/dWWpr8TtO0jSNf07xXZa/dpZSrnyLq6s3MdyyswLLKBvBGV24FDVrlWeiZ+pF7fwabZzXV3PFbWsCNLLPMwRI0UEszMeAAASSelU/DninSPGGi2usaDqtjrekXQLQX+nXCXEEwBIykiEq3II4PUGvgO58F+LfC/xZ+M/wAGLj4r+PfFHhy9+HDeI4rzVtVE2o290J9hSGbZ8kbhGUoq4KuRgnmu/wD+CV+maLZ/s26bcab461DxPqFxGp1DQ7vV0u4tCcSShYYoFGbYOPmKN9480ra69rkN2t6n2irbhTqZGML/APXp9BQUUUUAFFFFABRRRQAUUUUANavjn9ovxR40+K/7V3hP4EeHvGWqfD3w3J4fk8R63q/h+ZYNSuUEzRJBBKQTEQVB3L1DtkELg/YzdetfPn7RH7Kcnxf8b+F/H/hPxrffDf4keHo3tbTX7O0jvI5LZyS0M1vIVEi/M+PmA+dshuMZyXvK+w+jXUxfEHix/wBhv4XbNX8T+LPjDe6xrkWm+GdO1q4jl1Sa4mQBLZrkgb13I7eYy/KG24OBnN8P/tyano+r+M/D3xT+FWqfDrxZ4f8ADc/iyDS7fVINXTUbCEHeY5ogqh8gjbyPlJJGMVd8V/sba98RvhYmj+Nvi9rniHx5Za5D4j0rxithDarpt5CgSLybND5axgAkoGGWYtkGjwX+x34is/EXifxj43+L+seMPiRqmgv4d0/xNaaZBpX9j2zMXBggiLLvD/NuzzyO5q9dbkq1loeUeMP28fiL44/ZT8e+PPCHw+XwzqWlrata6taeIdP1a3gtptxa4JCFPPjKKr2rrvXzkJ9vqP8AZi8feM/iX8GPD+veO/DX/CM65c28TBPt8N39tiMUbLd5iULH5pZm8rGU6V4j4H/4J6Q6X4W+K9l4r8fS+ItZ+IVgthfahpWg2ujW8OwlklFrB8jS7zlmJBbnPJJPvvwB+G/iT4TfDLTfC/ifxofHd5p4EEGpnS4tPCWyqqRQiKMsDsVfvMSzZyTT0SYne6PSFr5W/wCCo/8AyYn8Tf8AuGf+nS0r6pWvlb/gqP8A8mJ/E3/uGf8Ap0tKCjyr/gir/wAms+Kv+xzuv/SGxr7/AK+AP+CKv/JrPir/ALHO6/8ASGxr7/oAKKKKAENfDH/BTP8A13w6+mo/+2tfc5r4Y/4KZ/674dfTUT/6S19Fw7FSzSjFq+v6Mwr/AMNs+rPgJ/yQ74e/9i/p/wD6Tx13lcH8Azn4G/D3/sXtP/8ASaOu8rxcR/Gn6v8AM1j8KCiiiucoKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAoor5g/bi/aF+LH7N+l+EfE3gDwBb+OfCiTXL+Ky0E0k1nbxrHIrK0T5hVo1ut07xyRoUTdjIVwD6forz74G/Hbwj+0R8N9O8a+C9Q+3aVdfu5YZAFubKcAF7edATslXcMjJBDKylkZWPoANAC0UUUAFFFFABRRRQAUUUUAFFFFACHimtTj3qNjzS6gfE/xM/aG+IPh/8AbGtfBFh4g+z+F21rSrVrD7HbtmKZbcyrvMZf5jI/O7I3cEYFfbagKDX5vfGPn/goNZev/CRaH/6Ba1+kCnOK+lzehTo0MJKEUnKF35vuc9KTk5X7jsVR1rQdO8SaXcabq1jbanp1wuya0vIVlilX0ZGBDD6irwpGYjpXzR0HFeHPgf8ADrwfp2p6foPgLwzolhqkLW9/a6bo9tbxXcTDDJKqIA6kEghsjFaOo/DHwhrHhCHwnf8AhbRb7wrCkcUWh3OnwyWKImNirAylAFwMADAwMVsx6xZTapPpiXlu+pW8EVzPZrKpmiikaRYpGTO5VdoZQrEAExuBkqcXFOaYzlvD/wAKfBXhOzv7TQ/COhaNa6hAlteQ6fpkECXMSIUSORUUB1VCVCtkAEgcVU8PfBP4eeEZLKTQvAnhrRZLF5pLVtO0i3tzA0qBJmQog2l1VVYjG4AA5ArtaKBHGS/Bf4fzeDY/CMngbw2/hSOTzk0JtJtzYrIWLFxBs2BtxJzjOST3qPTvgb8OdH0OPRNP8A+GLHRo75dTTTrXR7eO3W7UALcCNUCiVQoxJjcMDB4rt6KAMdPB2hR+JpPEaaNp6+IpLUWL6utrGLt7cNuEJlxvMYb5tucZ5xVTwl8OPCngH7f/AMIx4Z0fw4L+b7Rd/wBk2EVr9ol/vyeWo3t7nJro6KAEVdowKWiigAooooAKKKKACiiigAooooATFJtBp1FADdgo2inUUAN8sUuKWigBK+Vv+Co//JifxN/7hn/p0tK+qq+Vf+Co/wDyYn8Tf+4Z/wCnS0oA8q/4Iq/8ms+Kv+xzuv8A0hsa+/6+AP8Agir/AMms+Kv+xzuv/SGxr7/oAKKKKAEavkr/AIKQafbSfCHw7evbxNeQ64kMdwyAyJG9vOXUN1AYohIHB2LnoK+ta+bf+CgHhs65+z7c332nyf7F1G2vvL2bvO3MbfZnI2/8fG7PP3cY5yPZyWap5jQlLbmRjWV6bOk/YtvrnUf2afB0t3cTXUqpcwq8zl2CJcyoi5PZVVVA6AKAOBXt9fNv7AHib+3f2fLax+z+R/Yuo3Nh5m/d525hcbsY+XHn7cc/cznnA+kawzKDp42tBq3vP8yqbvBMWiiivNNAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACkZd1LRQB+dXx1+Bfi79gz4kX/AMe/gJp/27wDdfvPG3w6iJS3SBSWa4gVQdkS7nYFVJtizEBrdpI4/tP4F/HXwh+0T8OdO8aeCdR+3aVdfu5YZQFuLKdQDJbzoCdkqbhkZIIKspZGVj6CyhsZr81/2hf2KfiF+zV8ZYPjn+yzZ/8APWTWvBNvjyvLx5kqRQbl862l2DNsn7xJPLMA+4IAD9J1JZcmnV4n+yn+1d4R/ay+HMfiPw4/2LVbUpDrOgTyhrjTJ2BO0nA3xttYxygAOFPCurontanNAC0UUUAFFFFABRRRQAUUUUAJTWXmnGo5JNqk+gzQB+cPw1A8Rf8ABQW6Gqf8TPy/EWq7DefvdvkpceRjdnHl+XHs/u7FxjaMfpBjGK/OD9kEH4pftdan4t403yzqOvfY/wDXZ85jF5W/5fu/ac7tvOzGBnI/SBW3Zr6viT3cRSp/ywivwOXD/C35i1598fvjFpnwA+Dviz4g6vH9otdDsjOltll+0zswjgg3KjlPMmeOPeVIXfuPANd/uNfmx8erm5/4KBftp+HvhLosOoXXwj+Gd7JP4zuxDNDbyXqO6yQufOTdnyvssTqFlUyXcieZGu6vlDqO/wD+Cbfwe8R61eeLv2lviJJv8ZfEre+n2rLcodP00zFtoErnMcnl2/krhtkEEBWQiQqPupap6To9joOl2emaZZW+m6bZQpbWtnaRLFDBEihUjjRQAqqoACgAAACrgAXgDAoAWiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAK+Vf+Co//ACYn8Tf+4Z/6dLSvqqvlX/gqP/yYn8Tf+4Z/6dLSgDyr/gir/wAms+Kv+xzuv/SGxr7/AK+AP+CKv/JrPir/ALHO6/8ASGxr7/oAKKKKAGt615N+1Z4Z/wCEs/Z78c2f2n7IYdPa/wDM2b8/ZmFxsxkY3eVtz23ZwcYPrRrL8SaHZ+JvD+paPqMP2mw1C2ktbiHcy+ZE6lXXcpBGQSMggjPUVtQqexrQqdmmTJXTR8g/8E1fE1zN4Z8b6A0cQs7K8t7+Nwp8wvMjo4JzjAFvHgYBGW5PGPtFG3A1+b/7AeuS+Efj1qfh3VJrrTJtQ064tP7NmSRN93C6PtkTHyukaXHLgYyy5BbB/R9P0r6DiSn7PMpyW07S+9GGHf7tLsOpaKK+ZOkKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKTaCc0tFAHwp+1d+yn4u+GXxGf9oz9nNBY+PrXdL4j8JwRFrfxFAxDTMsKkb5WwC8QwZSokjK3Cgy+6/sr/tieA/2rfBtnf8Ah/ULfTvE6ws+p+FLq5U31kyFFkYLwZYN0ibZ1UK29QQj7kX3UqD+WK+C/wBrv/gn/wCINU+J9p8bv2etUt/BvxPs5pL+9sRKLeHUZgjHzYSVMazyk+XJHKBDMJSZCp8wygH3orbhS180/sZ/tmaZ+05oN9outWP/AAifxW8O7oPEXhW5V4pI3RvLeeFJPn8rf8rI2XhchHzlHk+lVORQAtFFFABRRRQAUUUUANauY+J3iK58H/DjxVr1mkUl3pmlXV7Ck4JjLxRM6hgCCRlecEH3FdQa8I/bU8b/APCF/s++IRFefY77VvL0u3/db/N8xv30fKkLmBZ+TjHYhsV1YOi6+Jp0kr3a/Mib5Ytnzj/wTa8Mm68eeLtfNxs+wadFY/ZvLzv8+Tfv3Z42/ZsbcHO/qNvP6Ahyq9M18m/8E5PCcem/CnXtdksJbe81PVDCtzIGC3FvDGoTYD8pUSPONyjk5BJ24H1Rq2rWPh/S7zUtTvLfTtOsoHubm8u5VihgiRSzyO7EKqqoJJJwACTXscQ1lVzKq1snb7tDKhHlpo+bP+ChH7VE37LfwHuNQ0S6t4fHOuTf2doSypHMYm6z3RiZwWWKPOG2uolkgDqVcg3f2B/2Ubf9lX4IWunXqeZ4z13ytT8QTSRQiSGcxKBZq8ZbfHB8yg73Bd5XXaJNo+av2XdOu/28v2xtY/aR1jS/s3w38HY0bwpp2oPbST/a4kDxmSMRnOz7RLdbi26OWaARySLE239KF9q+dOgWloooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAr5V/4Kj/APJifxN/7hn/AKdLSvqqvlX/AIKj/wDJifxN/wC4Z/6dLSgDyr/gir/yaz4q/wCxzuv/AEhsa+/6+AP+CKv/ACaz4q/7HO6/9IbGvv8AoAKKKKAEYUxu3FSU3bzS6gfmv4ijufgV+3ZHqF1LFHaXWvC9+26jEYIBa3xImkDFgNsYmlXzM7d0RJAwVH6TryAR0r4L/wCCk3g37P4g8H+Koo7p/tFtLplxLj9xGY3EkK5C8O/mz9TyI+B8rV9g/BXx9H8UPhX4Z8TI8Ty39mjXIgjZI0uF+SdFDc4WVXUcnOOpHJ+uzf8A2rBYXGdbcr9Y/wDAOWj7s5QO1FLSUtfJHUFFFFADWNG6qWt6xaeH9JvdTv5fIsbKB7m4l2ltkaKWZsAEnAB4AJrG+HPxH8PfFjwXpnizwrqH9q+H9TQyWl55MkXmKGKk7JFV15Uj5lHSgDp91Gaj30eZ8uSOKQDy1Lmoy3U/lRuPHHWn1AfupcmmZ/LtxSeYfTP0pAP3GjJpgk6HsahvL+3060nu7qeO1tII2llnmcIkaKCWZmPAAAJJPSgC1k0mayvDvijSPGGi22saDqtjrek3Slre/wBOuUuIJgCQSkiEq3II4PUGuQ+PHxu0T9nn4Xat488RWmoX2kaa0KzQaWkb3DGSVI12h3ReC4Jyw4z1oBa6I9ELYoya53xF4+0Hwh4NuPFev6nb6JoNrbC6uLy+cIkSEA8nPXJAwMkkgDJIrz74Qftd/CL48trS+CPGtrqzaNB9qv1mt57MwQjrKfPRMoO7DgdyKOtg6XPZM0tZHhvxVo/jDRbXWNA1Wx1rSLoFoL/TrhLiCYAkEpIhKsMgjIPY1qF6YDqKZu/OnL0oAWloooAKKKKACiiigApCoNLRQB8f/tmfsZ6p8QtesfjF8HL4+E/jp4e2z293busUetIibRBMW+TzdmY1eTKOhMMuYyrRav7Iv7evhn9oX7N4N8TRf8IV8ZbTz7fVPCd5BLDvmt8CZrcyD/eJgZvOj8uUEMsfmt9VY5zXyB+2Z/wT60v9ozXrDx94N1z/AIV/8VtM2yxa1boypfvEubYTNGQ8UqOsYW5Tc6ICCsm2MIAfXu406vj/APYz/bM1P4ia7f8Awd+MVh/wiXx18O7oLm0uUSJNaRF3GeEL8nm7MSMiZR0PnRZjLLFrfsL/ALSni/4z3nxb8GfEQafN458AeJp9Pu7vRbcx2EkEk06xJEWbe3lvb3CAsikxrAWLyGRqAPquikXkUtABRRRQAjV8Tf8ABSbxp5Wk+D/CUUlq/n3EuqXMe7NxH5a+XCcZ4RvNm6jkx8H5Wz9rs2Pyr80v2kbyb9oH9rqLwxpr7oYLm38Ox3NraSO8So5NzI6Egt5TvcZI2rsjznHzH6jhyipY5VpfDTTk/kjmrv3Ldz7V/ZS8Mnwj+zz4Gs/tP2rztPW/8zy9mPtLG42Yyfu+btz325wM4r5a/wCCkfxs1vxtr3hH9mb4Waxnxv4yvEh8QfYZXLWNg6DENz5cTskciu00pUhlgtyXUxzc/Yvxj+Lnh74C/C3XfHfit7hND0WBZZhaQmWaRmdY4o414+Z5HRASQoLAsyrlh8n/APBN34O+ItavPFv7S3xEk3+MviVufT7RhcodP04zFtoErnMUnl2/krhtkEEBWQiRlHztao61WVR7tt/edEVypI+tPgx8IPDvwI+F+geAvC0VxFoWiwtFAbuYyzSM7tJJK7dCzyO7kABQWIVVUADtgMUi9KdWIwooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAr5V/4Kj/8AJifxN/7hn/p0tK+qq+Vf+Co//JifxN/7hn/p0tKAPKv+CKv/ACaz4q/7HO6/9IbGvv8Ar4A/4Iq/8ms+Kv8Asc7r/wBIbGvv+gAooooASilooA8J/bW8EHxp+z34hMNn9rvdI2arb/vdnleU376TkgNiBp/lOc9gW2157/wTr+ID658M9Z8LTtK8ug3okgLRoI0t7jcwQMOWIkSdiWHG9RkjgfWkq7lx2Nfmj8Hmn/Zx/bM/4RxxdJp8movoZRvKmmmtbgg2jORgDJNrIxXDAAjHVD9Zly+u5diMF9qPvx+W/wCByVPcqRn30P0wU5zTqbGwZeDmnV8mdYUUUUAcb8ZOfhL42HT/AIkl7z/2wevzp+H934rv/gf+xl4T8N+ONc8DW/iSbVrK/u9Dn2SSQgOSNrZQsBu2MynYzB1GVFfp/f2NvqVrNa3cEdzazxtFLBMgdJEYEMrKeCCCQQeK52z+FfgvTodAhtPCOhWsXh9nfR0h02FBprPnebcBf3JbJzsxnPNSl71weqSPz61T4rfEz9n/AOGv7VvhzSPHHiPxnP4GvNKi0TWvEcv2/ULKK8CmeR5CMvsVsjI2grkKORUvxE8W2nw5/ZN+K178N/2lvE3xN1WHT9Gv3ubnW/tl3pM0l6iM0d1HholkAKm3JyoU5+8c/oXb+AvDVnfa5eQeH9LgvNeCjVriOyiWTUQqlF+0MFzLhSVG/OASOlYen/Av4baV4Z1Hw3ZfD/wvZ+H9SdZb7SbfRraO0unUgq0sQTY5BAILAkED0qltZ+QJ2d/U+BP2tNW+I/7Pvh/4Y+FbD4jeNfEFh4zvLzVdd16/8V2uh3RuFggK2ttqM0fk2EAIeQRY5yVBOaxLH4qfGKT9lHTFufH91ZamvxO07SNI8Qad4qstfu47KVc+Rd3Vm5juGQtlllA3grlduBX6ZeKPAPhnxxoQ0XxH4e0rX9HBVv7P1SyiubfK/dPlupXI7ccdqxtN+B/w50fQ49FsPAPhix0aO+XU00620a2jt1u1AC3AjCBRKAq4fG4YHPFHR36sd9F5Hw1b/CTx437R3xB+EY/aD+KH/CK2PhiHxQLltXVtTa6LbBGtzs3RQ7ssY4woYYB6bq8u8b/tTfFjxD+zL+zo0viDVreHxP8A2rFretWHiG28O3V29pM0cCHVJ0MVuSq7mLDMmCDknNfql/whXh4eILzXhoem/wBuXlsLK51L7JH9pntwciF5cbmjB52k49qyJvg14Bm8GweEJPBHhyTwlC/mR6C2k25sEYsWLLBs8sHczHIXqxPelfRII6O5+cVj8VPjFJ+yjpgufH91ZamvxO03SNI8Qad4rsdfu0spVz5F1dWjmO5ZWbLLKBvBGV24Fd/deCfFnhf4sfGf4Mz/ABX8e+KPDd78OW8SRXeraqJtRt7oTbCkM2z5I2CspRVAKuRgnmvtnTvgh8OdG0OPRNP8A+GLDRo75dTTTrXR7eO3W7UAJcCNUCiVQow+NwwMGt6PwhoUfieTxGui6eviKS1Fi+ri1jF21uG3CEy43GMN823OM84ptpt+aEuluh+U/wAOfHWhfCv9gbwxLpfxR8bX2veItXtrCbSdB8Y2dh/Y03+klbf7TMjDTIHVfMZiCSVU8KWI5iX4qeLPHH7GX7SeheIvEOqa9p+hanof9nf2v4jh8RXFust0pkX+0YFCXC5RSCvyjoADmv1eX4E/DZND1TRl+HvhVdH1WdbrUNPXRbYW95MDkSTR7Nsjg8hmBI9aE+BPw3h0XUNIj+H/AIXTSdRWBL2xTRrYQXKw/wCpEiBNriPA2bgduOMUJrW4lpa3med/tOeHvAfjD9mC70D4j+JP+EQ8M6rBZWrazv2/ZrgvG0DEkEbfMVc5wMZyV+8PlTw78TPEfjzWfiF8FtV8e+H/AIpQQ+BLq+074m/DzZbalbwxsCLO4eLfGu8ooKIeQRksW+X9GNU8O6XrmizaRqWm2mo6VMnlS2N1AksEicfKyMCpHA4IxxWX4L+GfhD4b2c1p4R8K6L4WtJm3ywaLp8NnHI3YssaqCfrStdy8xrRRXY/KDwv8Qta+D/7AXg+7+HfxF1rVdS8Qa7Z6b4jtV8SQJ/wjkDG5Igt5HUrpnnbVJkk6H5uldH4V+LvxY039l39oOWXxrfzaZpVnp91oeoD4iaf4l1fTpnuAsqvfWLh1V9pK7lUAAqufmr9L7H4NeANL0bWNIs/A/hy00nWX83U7C30m3SC+f8AvTxhNsh92Brzv4q/si+CfGHwZ8Z+APBui6B8Nj4nghhutR0LQ4IyfLkDoXjiMXmY+YAFhjefxTbs/MceifQ+VPhL8UL/AE/9pz4LQeDPj5rXxjTxvp91N4u0e+1CG5tdP2WwkWSOBFC2Z37v3eA/yYJw2D+kCniuJ8B/Bzwf8PJDfaL4Y0PTtdntkt77WLDTIba5vdoGTLIq7nyw3fMx5rtlGO9aNrYhLW44UtJS1JQUUUUAFFFFABRRRQAVyXxX8H6n8QPhz4i8OaN4o1DwVqmp2Ultba/pao1xZOwxvUN+R2lXwTseN9rr1tJigD8V/wBqn4O/tQeE/C1nqXxK8L/8JtfeC/JbQvix4TuidTsY4kSbFxJGBPJbQrFO/nTwpIk7GQXBVikx/wAE2/2gNT8Yf8FAm1rW9J0+TWvHei3WmX0+mwrZRC4it47qS8aJFKtLM1gWkC7AZLh3GMbD+0+0V5XD+yx8KbT4x2fxVtPBdhp3j6188pq+ntJbeY8wlE0ssMbLFLK4nl3SOjOdwy3yrgA9UU5HFDMR0Ga81/aA/aG8Ffsy+A28W+OL64tNOeb7JaxWls9xNd3JiklSBAowGdYXAZyqAgbmXOa/I34yf8Ff/jV4416OXwOdP+G2iw52WlvbQ6lcTbkQHzpriIq2GVyvlxx4EhDbyoagD9uzJ37U5TntivxC+Dv/AAV++NXgjXJJvHB0/wCJWjTYD2dxbw6bcw4RwPJmt4gq5ZkLeZHJkR4XYSWr9XP2af2qvAX7Vng251/wTd3CPZz/AGe/0jUkSK+smJbYZY1ZhtdVLI6sykBhncjqoB6R4z8SW3g3wprOv3iSyWml2U19NHAAZGSNC7BQSAWIHGSBnuK+Bf2DtEvPiF8evEPjbWIf7Qlsrea7kvg4j2X1y5G7y0IB3p9p4C7R7HbXsn/BQ34lDw38L7DwnFDvuPElx88rLlUht2jkbaQwIcyGHHBG3f0OK8W1r4v2X7C/7EEHiOG38v4lePzKNIVZbYzRM8b+RdkMGLwQReXKUKv+9nVG2eaSv12GX1LJ6td/FWfKvRas5Je/WS7En7RFxqf7dH7Y2h/A3S4ft/we+Ht7Bq3jW/ghV4nvVSXNsbhJgy5VjahEKSpI90xRxACn6EaRo9joOl2emabZwadp1lClta2dpEsUMESKFSONFACqqgAKAAAABxXzv+wP+ynb/srfA+0069Tf4z1zy9S1+aSOHfDOY1As1eMtuigyyg72Bd5XXaJNo+lK+ROsAAvAGBS0UUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFfKv/AAVH/wCTE/ib/wBwz/06WlfVVfKv/BUf/kxP4m/9wz/06WlAHlX/AARV/wCTWfFX/Y53X/pDY19/18Af8EVf+TWfFX/Y53X/AKQ2Nff9ABRRRQAUUUUANb1r4Z/4KQfDjbJ4Z8fRTYBxolxC79/3k0TIu3/rvuJbtGAOpr7mNcf8XPh7ZfFX4da74Vv28uLUbYxpNhj5MoIaKTCspbZIqNtyA23B4Jr1crxn1DGQr9Fv6PRmVWHtINHLfsv/ABePxm+EGk6zcSb9Ytf9A1T5cZuY1XL8Kq/OrJJhRtXzNoJ2mvWVJavzm/YV+Ktz8N/ilfeAtfaaws9akaBbe8Jj+y6jHkBSrsoQyANGQFLs6wrjg1+jCcLW+dYJYHGShH4HrH0YqU+eOu46lpBS14ZsIy7qNtLRQAm2jFLRQAmKMUtFACbaNtLRQA3bRtp1FACbaNtLRQAlG2looATbSbBTqKAG7RS7aWigBBxS0UUAFFFFABRRRQAUUUUAFFFFABTWbHFOpr/TNAH4Hf8ABTj9oK++N37UXiDTFa4g8PeC5pvDun2cpZQJYpCt3OU8xk3STKwDqFLRRQBgCtfJROa+gP28vhnrfwx/a2+JdprcHl/2vrNzr1jPGjiK4tbuZ542RnVd+3e0bFcqJIpFBO3NeAMNpxQAq19Z/wDBMr4/33wR/af0DTWa4m8O+NJovD2oWcJZh5ssgW0nCeYq7o5mUF2DFYpZwqksK+S1rsfhFHet8SvDk2nXL2d9aX0V5DcRyNG8bQnzQysvKsNnykd8cjrXVh6MsTVhRju3YUmoq7P00+J3iC2/aR/aqnh1HWLLTPClnPLZnUmvI0t4NLs1klnnE+zbhkjnlDOCqmQAsVGa6X9mfR7r9tD9rbUvjJcLC/wS+GtxLongTSC0f2cXUccQSZLVoVKDyytzllSRHe1RWcQEJ8x3WoeIL/TNP+FPga3kbx98V5BpBvGS8i+waMsyGWVWiBWSGeSKZZnw3lRWNwGUiXK/sD8F/hD4e+Bfwv0DwH4VjuI9C0WFooDdTGWaRmdpJZHbAG55HdyAFUFiFVVAUe7n1aKrRwdH4KS5fn1f3nNQi7Ob3Z2y4xx0p1IvHvS18wdQUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFfKv/BUf/kxP4m/9wz/ANOlpX1VXyr/AMFR/wDkxP4m/wDcM/8ATpaUAeVf8EVf+TWfFX/Y53X/AKQ2Nff9fAH/AARV/wCTWfFX/Y53X/pDY19/0AFFFFABRRRQAhGaTaM5p1FAH59/t7fCL/hX/jLR/iR4aj/smPUrjZeS2R8lotQUmRJ1IfIeRVY5RRhoSxJaTn69+AvxYt/jN8LdG8SQmJb2SPyb+3jIAhuk4kXbuYqpPzqGO7Y6E9a1vi18PLL4rfD/AF3wpft5UOpWxjSfBPkygh4pNoZS2yRUbbkA7cHgmvgb9mLx1c/sy/H7WfB/jCePTNMvZTp2oTzZjhimjLG3udzR7vKbJAY7F2XAkY4UV9lTX9r5Y6X/AC9oXa7uH/AOP+FUv0kfpRGxZeadTI23Lmn18adgUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFfKv7Yf7VnxD/Zb8VeFteh+Gv8AwkXwZj8tfE/iK3nD3lvJNI8aRwxBx5ezajF5VMcrTRxB4mIY/VVUtY0qx13S7zTdTs7fUNNvIXt7q0u4llhnidSrxujAhlKkgqRggkGgDL8A/EDw/wDFDwbpPivwrqtvrfh/VYRPaX1sTtkXJBBBAKsrAqyMAysrKwDAgdApzX5meP8A4R/GP/gnD4z1bxv8Eorjxh8AppjrWv8Ag25lWVtOVQI5VDPmUKEYMtxEGZVhX7SsiQBpPvL4F/HTwj+0R8ONO8a+CtR+3aTd5jlhlAW4spwAXt50BOyVdwyMkEFWUsjKxAPQqKQUtABRRRQAUUUUAFFFFABRRRQAUhGaWms23tmgD5A/4KDfsI/8Nd6Hoeq+HNQ0/Q/Huh7oILnUI9tvfWrsC0M8saNKPLIZ4yAygvKu396XT8WvjJ8AfiF8ANeTSfiD4U1Dw1dS58iS4QPb3OFRm8mdC0Uu0Spu2M20ttbB4r9m/wDgo3+3FN+yn4O0/QPCi29x8RfEkErWk0rxyLpNupCm7khJJZmYssSsvls0chYsIzG/Zf8ABPz4jfFX4sfs46f4m+LlvjWr28mk0y+e1jtZdQ05lRobh4o8KuWaVVIRN0aRuA27e4M/Cj4X/BXx38aNSOn+BvCOseKZ0mht5n020eSG1aZisZnlx5cKsVb55GVQFYk4Umvb/DPw40z4CvrU3iTV7fUEtro213qmgoLyLyUl2brUt5fmh+GGWRWymSoG6v1s/bY+PP8Awq34enQtE1X7L4v1rCRfZ5ds9ra5PmTj5Tt3bTGpyrZdmQ5jOPy8+FHwfuP2wP2kNJ+F0N5LY+F9NifVPEF/Z+S08MMQCt5ZdupaWKEYDFGlLMjqhFfZ5TThgMJUzKtu9ILz7/I46kvaSVJbdT7Y/wCCWn7Od5oWk698efE9rFYa548iZdI0uNbpP7O0wzFyP3zkukxjgaPdvIiiiYSN5rAff61S0fR7HQdKstN0yyt9N06yhS2trO0iWKGCJFCpHGigBVVQAFGAAAKugBeAMCvjpydSTlJ3bOpaaC0UUVIwooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAK+Vf+Co/wDyYn8Tf+4Z/wCnS0r6qr5V/wCCo/8AyYn8Tf8AuGf+nS0oA8q/4Iq/8ms+Kv8Asc7r/wBIbGvv+vgD/gir/wAms+Kv+xzuv/SGxr7/AKACiiigAooooAKKKKAGtivjv9vX9n1PEWgv8RdBtY49U0yP/ibxwwuZLyDKqspK5G6EZySP9XklgIlFfYp71HIobAIyD1ruwOMqYCvHEU91+K6oicFNWZ80fsSftCSfFLwbJ4Z1u6kufFOhRjfc3UyNJe27MQkmOGLJ8qOSDnKMWLSED6ZV91fmV8bvAOr/ALHvx20rxN4TWOHSJpHutFFxJ9owoQJcW0gOGwPNKg/e2SJh94Yj9APhD8VNJ+MHgHTPEukywt9ojVbq1jkLm0uQoMkDkhTuUnqVG4YYfKwJ9jOcFCPLjsL/AAqmvo+qMKNRtOE90duKWmo24Zp1fMnUFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFIVB60tFADdgHTivzt+Ov7GfxL/Zu+I+o/GX9lm/+w2H/IS1z4bRtIbe+eMkGOC2T5J4iks7CAlXiO77OdzxpH+ilJjNAHin7KP7V3hD9rL4cR+I/Db/AGHVbXZDrOgTyBrjTJyDhScDfE21jHKAA4U8K6uie1qc18Q/tRfsIeIbjx9qXxn/AGevFFx8PvieYWuL7SbJhDa+IJlljlIZiwRGkKZdJFeGZ1jMgQmSRvSf2NP2zNN/ab0G/wBG1qx/4RL4reHd0HiLwrcI8To6P5bzwI/z+Vv+Vkb54XOx85R5AD6WopFO4ZpaACiiigAooooAKaxPanV5n+0x4/m+Fv7PfxH8V2mrW+h6lpWgXs+n31y0YWO88lhbACTKMzTGJVQg7mZVwc4IB5/+zn+3l8L/ANqLx74k8I+Cn1galo0DXiS6lY+TDf2qyiJp4CGZgoZ4vllWN8Sr8vDhdT9sL9qrw9+yn8I7/wAQajd27eKL2Ca38O6TIhla9vQnylow6nyI2ZGlfcuFIAJd0VvzE/4JaePvAXwDt/jH8Y/HOrXFgmgaZY6NZ2kJRnvmvJJZTDFEQGknLWEe3DKqqZWfCgunP+BfC/i//gqh+2Rfa5rdtqGieEVCPqU2nyG4i0PTY0YQW0bykKJJWVh8q8ySzTCIqrqADf8A2J/2SvEP7enxP1z4tfFfW7jU/CdvqmzVZmuAt3rN4qI4tFCEGCBI3iBKhdqFI4gOWh/ZHx5400z4d+DtW8SaxL5Wn6ZbtcSAMqtJgcRpuIUuxwqgkZZgO9N8BeAfDvws8F6V4U8K6Tb6H4e0qDyLSwtgdiLkkkk5LMzFmZ2JZ2ZmYkkmvz//AG1P2h7b4t+KLPwj4c1Syk8J6XMJJL+Z44YJ7350Mv2iRgohRHIDkqnLsSyhGHsZXl8syxCp7QWsn2XUxq1FTjc+Xv2p/wBoTVPE2sav4s1OeZ9T1CR4NMtJJlf7FF8xjRcgApECMkJ8zHJwXJr9Jf8Agnx+yxL+y78CYLDXbSCHxzrsx1DXXieOUxt92G2EqqCyxR4yu51Esk5RirAn4u/YG+C0X7SH7Q+oeOtRmvr74Y/De9jk8Pxi7nW1u9ZHlH7RFugiyn+ji4eNtkqhrNJPMUHP63iurO8xjjaqpUValDSK8hUabhG73Y4cClpBS186bhRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFfKv/BUf/kxP4m/9wz/06WlfVVfKv/BUf/kxP4m/9wz/ANOlpQB5V/wRV/5NZ8Vf9jndf+kNjX3/AF8Af8EVf+TWfFX/AGOd1/6Q2Nff9ABRRRQAUUUUAFFFFACYpNtOooA4/wCKXwv0D4u+Ebvw54itPtNlN88cqkLNbygELLE2DtdcnnBBBIIKkg/nh4D8WeIP2JPjxqunaxZXV7o02ba5iwYvttn5h8m8hUOULgAkBidu6WMlGJK/p6wzXj37TXwBt/j54DGmxzQ2OvWMhudOvpowVD7SGidsFhE/G7byCqNhtm0/R5TmEMPzYXFa0amj8vNehz1ablacfiR6h4f8QWXibQ9O1jTZvtOnahbx3VtNsZd8TqGRsMARkEHBAPtWgrEjnr6V+cP7PH7RHiD9mTxhdeAPiDbXUHh6K4Mc8Ew3zaTKxyZY8Z3wtuDFVyCG8xMkkSfopp+oWuqWFve2VxFd2dxGs0NxA4eOVGAKsrDgggggjqOlceZ5bUy6ryt3hLWMls0XTqKotNy5S01TkdMU6vINQooooAKKKKACiiigAooooAKKKKACiiigAooooARl3V8f/tm/8E+dL/aO16x8e+DNbHw/+K2m7ZYtat0ZUv3iXNsJmjIeORHWMLcpudEBBWTbGE+waQruoA+QP2M/2ztU+ImuX3wd+MVh/wAIl8dPDu6C5tLhFij1pETcZ4QvyebsxIyR5R0ImizGWWL6/UlhkjFfOn7XX7DvgH9rvS7STXGuNB8V6dDJFp/iLTVQzKrKxWGdWGJoBIwk2ZVgdwR03vu8W/Zo/bE8Q/BXxlc/AX9p+/t9A8W6RDv0bxxqVyEsdbswDsaW5k2gsVRtk7YMm1kkC3CESgH3oTQK8U/bSuvGFj+yr8Tr7wLrX/CO+JLHRpb6PUlco8UEJEtyI3CsVla3SZUYYKuykMhAdV/Yv+MWp/Hz9mDwB441uPZrWoWTwX0m5T9ont5pLaSfCIir5rQmTYqgJv2jO3JAPa6KKKACvjX/AIK1eNbHwr+xf4g0y7iuHn8Sanp+lWjQqpVJVnF2WkywIXy7SQZAY7mXjBJH2Tnmvir4z/tbfsv/AB4+JGi/AHxYn/CwrXW720SDVdKAn0u3v5Cfs6JdwSiVZSWWMvACoE5R3A80KAfi3pnjLW9V8K6X8PrnxINK8Gf2ydUMF0H+xwXUsccD3cyxI8j7YowBhXZV37FzI279o/2Q/wBpT9lX4W/DfSPht4L+JWgWcmm2YudR1HVLW40VNSusRpPdPJeKqmWRsER+YzKihV+SIbbfjb/gkz+zt4p0uK00zw/rHg6dJlka+0XWZ5JnUKw8si7M6bSWByEDZUfMBuB+K/2uv2Efg38GDY6F4J8Y+KLzxn5yy3sOqS211b21sUbCsIoYikrMY2UbjhAxIG5CevC4WtjKio0I3kyZSUVdn13+1h+2t4f1nw9f+C/hvrOm+I47+FrbVdd0+4jurWOJ0G63idSVeRkYbmzhA2BlzmP5N+JXgbxH4Ws/CPwc0jRdnxi+KuI2lne6EugaI7qm5kij+Qz7Lrz2y5jtoZUdF81sdl+yf+zLof7Lvw31f9oD4q3t+vhfSfs+padY2tsVuLyRZAts4jByFaWRAm5lVyys5SJSZPY/+CbvwU1rxrrni39pj4p6Op8beMrxp/D/ANuicmxsHU5mtvMldkikV1hiDKGWCBdjNHNz7mJxscJhFl+HVm/jfd9vRGMY80/aP5H2B8Afg7pfwC+DfhP4f6RL9otdDshBJc4dftM7EyTz7WdynmSvJJs3ELv2jgCvQNtCd+O9Or5g6BKWiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACvlX/gqP8A8mJ/E3/uGf8Ap0tK+qq+Vf8AgqP/AMmJ/E3/ALhn/p0tKAPKv+CKv/JrPir/ALHO6/8ASGxr7/r4A/4Iq/8AJrPir/sc7r/0hsa+/wCgAooooAKKKKACiiigAooooATHOaQqDTqKAPD/ANpr9mXSvj1oAmgMWm+LrGMiw1NgdrDJPkTYGWjJJ55KEllByyv8m/s7/tD6/wDsyeMLn4f+P7W6g8OxXBimtpVLzaTIxz5sYXO+Fs7iq5BDeYmSSJP0gYdPSvGP2i/2Y9C/aA062eWb+xvEVnhbbWIofNYR7stFIm5fMTliBkFWOQcFlf6XLszpqk8BjlzUX98X3X+RzVKb5ueG567puqWurafbXtjcQ3lncxrNDPbuHjlRgCrKwOGBByCODmranNfmT8NfiT42/Yp+KV34Z8TWctzoM0oe906Jt0cyH5VvLRjgFiFx2DbSj7WUGP8AQf4YfFDQPi94PtPEfhy8+0WU3yyRvgTW0oALRSqCdrjI4yQQQwJVgTy5llVXAWqRfPSltJbP/JlU6nPo9GdfS0yM5z9afXiG4UUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAcn8VvGGp/D/wCHPiLxHovhfUPGuq6ZZSXNtoGlsq3F66jhELH8TtDOQDsSR9qN+a3xc/ba/Zz/AGuNB0/wF8evBfi/4YeJNNvLhZdTggSd9AnjZg8QlCGc+YIkjkja1IDkZXMSyr+qmK5/xr8O/CvxK0uLTPF3hrSPFOmxTC5js9asIryFJQrKJFSRWAYK7gMBnDEdzQB+W37Kv7d3hn4IrefBj4q+LNP+JPwcnspovD3jBLOS422J8yP7FfWTK06xsEZBFIhaPIA822eKRPYP+CNvjqaX4R/ED4c391cXmpeEPEBlDreR3dhFb3KFVjtZY5GRl861upDs/dnzg6s29iPSviZ/wSn/AGefiF9oltPDmoeCr+4vGvJrzwzqDx7t2/dEsMwlgjiywIWONduxQu1cqeU/Yr/Yy8d/se/tNfECzsb7+2PgzrujLNZ6lK8AuDeR3C/Z4J04kEkcUl2C8YETh1Y7WIjjAOs/bC/4KL6J+yJ8UvC3g2/8G6h4k/tKyj1TUL22vEg+yWrzvEphRlbz5f3MxKM0S8IN/wA52fTnj74g+H/hb4P1bxX4r1W30Tw9pUBuLy+uSdka5AAAAJZmYhVRQWZmVVBLAH8r/wDgt74KsdP8ffCzxcktwdS1XTL3SpomZfJWK1ljljZRjcGLXsu4kkEKmAMEnxVNW/aI/wCCqfxOudMS9t7fR9NhiuZrJZZ7Tw7pBRHWORkzKxnlZpQrESStvcDEUZ8sA9V/ao/4KAePP2vfGV58E/gVpdw3hjXZl02K6tonj1PXFAfzslmAt7R15YMFby42aV0R5Il+h/2G/wDgmHofwV/4R74g/EpP7Z+I9vm7h0fekmnaPIdpibAB865iwx8zcY1Z/kUtGkx9+/ZX/Yt+Hn7Jegp/wj9l/aXi26sktdW8U3mftF7hi7BELFYIyxH7tOoji3tIyB68f/ac/bktmsdR8J/De5m+1NI1tdeJIyFjCYG77IwJJJJK+bgAbSU3bldfRwOAr5jVVKhG/d9F5szqVI01eR6P+1n+1pbfCGxl8NeGJorzxtcR/M+A8emIwyJHB4aUg5SM8YIdht2q/jn7Jv7J9z8Qb+L4j/EaOa7024k+2WVhfkvJqbsd32i4LZLREncA3MpO4/J/rNX9mX9hkt/ZPi74jRfL/r4fC80PJ6GNrok/7xMGP7m8/fjr1z9uj9p6w/Zd+A+sarbajb23jXVYJLHw3Zs6mZ7lgFa4VCjhlgVxK29dhKojMDIufexOMoZXRlgsvlzSfxz7+S8jnjB1Xz1V8j5s/al1Rf28v2svD/7NehXlxZ+CPBc8ut+MNZsZrWUvLHGqBIMklWiac2xOSVluXLwkW+W/RPR9GsdB0uz0zTLODTtNsoUtrWztI1ihgiRQqRoigBVVQAFHAAAHHFfO/wCwT+zZqf7PPwblk8YE3fxM8VXsmteJ7+a4W6ned2OyF7gKGk2KdzZaQedLcMrsr5P0qOBivkDsBVCjA6UtFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABXyr/AMFR/wDkxP4m/wDcM/8ATpaV9VV8q/8ABUf/AJMT+Jv/AHDP/TpaUAeVf8EVf+TWfFX/AGOd1/6Q2Nff9fAH/BFX/k1nxV/2Od1/6Q2Nff8AQAUUUUAFFFFABRRRQAUUUUAFFFFACUjIGp1FAHD/ABU+EPhb4v8Ah+bSfEulxXm6Jo4L1VUXVoWKktDIQShyiE9m2gMGGQfz48WeA/ih+xL4vstZ0/VDPo1zcfJd2pk+xXuwvthu4SQA5QswUk43MY3LKWX9PHGarahptprFhcWV9bw3lncRtFNbzxh45EYEMrKeGBBIIPHNe5l+bVcCnSmuelLeL2+XZmM6Sm7rR9zwz4GftkeDPi/FBZXs8PhbxNLIYxpV7NlZcuqp5UxVVkZt6gJw+QwCkAMffFbNfE/x2/4J9Wc1s+q/DF/stwm5pdD1C5LROojGBBKwLByy9JWKkyffjC4PAfCj9tDx58Gb5fC3xC0vUNctreXMx1TzItXthIUbJMnMgCFmVHAJ3qPMVQAPTqZTQzCHtsqlfvB/EvTujFVZU3ar95+jG6lrzr4U/H7wT8ZrFZfDWswzXgj8ybTLj91dwYCbt0Z5IUyKpdNybuAxr0NTmvlKlKpRk4VItNdGdaaauhwpaRaWsxhRRRQAUUUUAFFFFABRRRQAUVxPxo+L3h74C/DHX/HnimWePQ9GhWWYWsJlmkZnWOKNF4BZ5HRBuIUFgWZVBYc9+zb+0p4R/ao+HB8Z+DBfw2Ed7Lp9zaapbiK4trhArFHCsyHKSRuCjsMOASGDKAD1VmxXwb8Vv+CifinwB+3zonwQtPCuj3PhOXU9K0e7vJpZft0kt9HEwnjcHYixm5i/dmNy3lP867x5f0l4L/a6+DfxJ+J0vw98MfEHR9e8WJCZ0tLF2khnUIsjCC4C+TOyo24pG7MAr5A8t9v43/tu/F2HRf8Agot4t8eeFpLfU5/DWv6dLCLyGQQteafDbRyRSL8rFVnt3Q4I3BSVbBBoA/eTVtYstB0u71TU7y307TbKF7m6vLuQRQ28SKWeR3YgKqqCSxIAAJNfFnx2/wCCtfwd+FtvcWng+S4+J3iGKaSA2+mFraxjaORFYvdyIVZWUuyPAsyt5fJUMrV8L2fgr9q3/gppqkep6hLcReCGmaa1uNRZtN8O2pDXITyI1UtcMjebB5qLNKuVWVwOa+zvhP8A8Eqfgd8Cbe08UfEfW5fGl1p+Gml16WOw0ZZfPVopDbg5PASMpLLJG+9sp8yhXGLk7IOh534B1bxJ/wAFav2c/H2neMfCegeHPF3he8gbwp4utLS6S0W4kXdNa5Z3YfLEiy4dxtuYH8otFHv+ZPHH7Lv7V37Hug6rHp3iXVdF8HxXCTG78I+LzbWt7cSRpkxW3mxTyS4Xaf3O4iEkZVQ1foF8Rv2+vDHgvSrbw/8ACrQLe8trGGO2trm4tza6dbRIseyOG3Xa5ULujA/dhCqkBlryb4e/s2/FH9qXU38VeLdYutO06X7uqa1HI8s0bq0q/ZYPlUw7nHQpGBIdm7aVH1GHySUaf1jHy9lT8/ifotzmlW15aerPjXxl+39+0hq2hyaR4y8S3Vxpt5keTcaZFp3mrtZXQtbpCZEIkwyMWXpxxX6k/sa/sp6b4P8ABmieNfGOjrP4y1S1t79NP1G3JOi7lSVY9kigi4VsbnKhlK7Vxhi/unwj+CfhP4K6HLp3hjT/ALMbjY13eTMZLi6dV2hnc/idqgICzEKNxruz8vHSvLnjp0YTw2Fm1Tfpd+tjRQUrSktRu0ZwB71+cnwSkf8A4KGftlan8V9Wt7j/AIVJ8LZls/CMUNxdRQ39+s/mxXZDoqliqLNJGvlso+wpIJFDFvQP+Cj/AMXtW1628N/s2+AJLg/EX4kTQRXLxw3Ajs9IaR1lkeWLkK7QuJMLIBbx3O9VBQt9PfAH4O6X8Avg14T+H+kSefaaHZCCS52sv2mdmMk8+1ncp5kzySbAxC79o4AryjU9BXOOaWkpaACiiigBDSAmvPP2hPi3/wAKJ+Dvinx3/Zf9tnQ7T7T/AGf9o8jz/mVdvmbW29eu09Oleffs8/thaL8avgn4j+IGr6WfBknhma7h13Sprk3LWPkJ5hO/Ym8GMhuF65HOKm9726B28z6E3Gjca+LPhf8A8FEdS+KHwj+IXjO3+Gtno134Vm09IdN1nxbbWMV5HdkFJHuriKOKD5GDAHduyADkjP0xq3xv8BeF4bg+IvG3hnw/PaJb/bYdQ1q2hNq0yFolcs4xvCsVzjcFJGQKqzsLyO7z3pNx9K8u8V/GmTQ/iR8N/DumWGkaxpHi9bmQ6u3iO1tpIY44hIj29s58y8D558n7gwx4NbNr8dfhxeajpWn2/wAQPC09/qxZdPtYtatmlvCrtGwhQSZkw6Op2g4ZWHUEUDO53Uua5LQviv4L8U2us3WjeL9A1e20UsNTmsdTgnSxKhiwnKuRGQFYndjG0+hqU/E7wgNN0LUP+Eq0T7Br0qW+kXX9owiLUZHBKJbvuxKzAHAQknHFAHT7qN3WuG1b46fDjQLg2+p/EDwtp1wL59MMV3rVtEwukCl7fDOP3qh0yn3hvXI5FamkfEnwn4g8Sal4d0vxPo2peINMGb/SbPUIZbq15AzLErF06j7wHUUBsdLuNG4141+0h+0xpX7PGk6Ej6Nf+KvFfiK8/s7QvDel7ftF/PgZ5b7kallDPg43rwc1N8Ffi9458bya1a/EX4U33wrvdOijuI2n1aDU7O5icuCVuIgFDrs+ZCMgMp71MXzXt0B6WR6/uo3Vx3g34weA/iLfXln4T8a+HfE95ZqGubfRtVgu5IASRl1jdioyCOfQ1max+0F8OtI0XxVqP/Ca+H75fC8DT6tb2Wq20ktpjICSL5nyMzDaofGWIFF7LUFqei5oFeYfAL9ofwb+0h4HtvE/g/UFlikXNxps80JvbElmCpcRRSP5TNtLAE8jmvTl6cHNVqhJ3HUUUUDCiiigAooooAKKKKACiiigAooooAKKKKACvlX/AIKj/wDJifxN/wC4Z/6dLSvqqvlX/gqP/wAmJ/E3/uGf+nS0oA8q/wCCKv8Ayaz4q/7HO6/9IbGvv+vgD/gir/yaz4q/7HO6/wDSGxr7/oAKKKKACiiigAooooAKKKKACiiigAooooATFG0UtFADWUHmuJ+KXwg8LfGDw7LpHiTTIrrMbJb3qIourNmKkvDJglDlFJ/hbbhgy5B7em7d3XpVQqSpTU4NprqhNJqzPzt+JX7BXjrwPrUOq/D3Uf8AhILeO5EttidbO/s2DMyNuZlRtgWP94rKxY5CADNO+Gf7bvxA+Eupp4b+JGkXWs29sY45ft0TW2q2yFY8FtwHm/uwWAkAdzICZMV+h+0denNcn8QvhT4R+KmmrYeKtCtdYhTPlSSArNDllY+XKpDx5KLnawyBg5HFfVQz5YqCpZnSVRd9pL5nM6PLrTdjhvhn+158NPijc2llZa3/AGTq9zkJpmsJ9nlLeYEVFfJjd2LKVRHZiG6cED2USH618YfFD/gnNpd5HJdeAddl067Ls/8AZ+tMZLbazrtRJVXeiou/7wlLfKCRyx8ub4c/tK/s5/uNAm1bUNIj/wBFgXRX/tO0/efvWKWjqzJ8wOZDEuDuAbDjc3lmX4zXA4hRf8s9H9+wvazh8cfuP0hD8Cl3cV+d3g3/AIKNeNNI8uLxF4e0vxBBHbiPfas9ncSSjA8x2+dDnDZVY1GTkYAwfa/Cv/BRD4c6xNY2+r2GtaBNLGDc3Etuk9tbuEyw3RuZGXI2g+WCcgkKM4463D+Y0Vzez5l3jqvwNI16cup9T5oryXwz+1Z8JvFgufsXjnS7f7Pt3/2mzWGd2cbfPCb/ALpztzjjOMjPVaL8XvA/iPUodO0nxl4f1PUJt3lWlnqkE0r4UsdqKxJwoJOOgBNeNPDV6fxwat5M1UovqdfmlyaiWZMcsPzo86PswP41hZ9hkvNFc14++Ivhz4X+D9W8VeKdXttF8P6VCZ7u+uW+WNcgAADLMzMQqooLMzBVBJAPn2j/ALY3wa8QeBofF2n+P9LudGm3bAPMS7OJTEf9EZROPmB6x/d+b7vNVClUqS5YRbYNpbnz5/wWOOt/8Ml2f9lf2h9h/wCEmsv7W+xb/K+y+VcbftG3jyvtH2bG/wCXzPK/i21+POheJNW8XaVo3gTU/GVv4Y8GwTvdCO+WddNjuNsrfaJorSGR5Z2DmITGN5ApjQssaAL+8XiT9vT4SaLZxz2Wp6hr8zSCNrbTrCVHUYJ3kziNcDAHBLZYcYyR+eX7SGh/AP4sXWozeDPhRP4H1GWaCWLWdL1AWiMqRhHjbT1V7dEPrEVYsquW+Z1b2sPkmY4j4KLt3en5mUq0I9T6a/4J9/sL/CHwDqNh8XfDHxL/AOFs6rbmWPT9QsEWzs7DzrdAyS2wd5FuRHK4ZZXG1ZwDErKGrzT/AIKRfsXx2PxIi+Mnw+8U+HfAmqyQtqF3prXU1nf3uqQyB1urLyg2Z5C8e4gRKjoJWcmR3X5c+BP7I/xXl+I2h+IPhvLqsv2G4Yf2zpiy2ZicRkyRfaAwRN0bbTukUkSEYOQG+4vh/wD8E6PFWo31jN4v1rT9G0x40mnttOc3F4CSpaE5URodpcbw0gDAYVxzXTRyvD0pSePrKFui1b9CJVW/4aufGfgH/goh+0z+z/EtjrOqf8JdpcEbW8cXjC2N6okdxLv+2Rsk0jj5gA8zKFJG35V22PDf7Tmu/tXfFLw74e8V63cQa7rN8un6fLqTs+nwzShEjUJDH+6MjrGhKRY3FWcgbmH60fDD9kj4a/Cu4tL6x0U6rq9tympavJ9olDCQSK4TAjR1IUK6IrAL15JMdr+xf8DrD4jaJ45sPhloGmeJNF2tYy6dbfZbeJ1LFJTaxlYHlUuSsjRl1KoQQUQrzwx8MvxDqZft0ckrlOHtI2mY/wACf2MfB3whuE1W+b/hLPES7Sl7f26CK3ZZN6vBF82x+E+cszAp8pUMQfoLyxSrxTq8vE4qvjKnta8nKT7mkYxirRQm2uS+K/xK0P4N/DnxF428Rz+Rouh2Ul7cbXRZJNo+WKPeyqZZGKxopYbndV7111eVftJ/s1+EP2qPhyPBvjM6hDYx3kWoW13pdwIbm2nQModCyshykkiEOrDDkgBgrDkLPl7/AIJ3fCfxB8S/Eviv9qD4taNbr468YTIPD4kshALHTxCsYnt13kqsseyFGdfMMUO7e63DFvvRelVNJ0ax0HS7PTNMs4NO02yhS2tbO0iWKGCJFCpHGigBFVQAFAAAAFXAAvAGBTAWiiigAooooA8J/bi8O6r4s/ZV+I2k6Hpl5rOrXWmGO3sdPt3nnmbzEO1I0BZjgHgCvjr4u/s6/EyT4l6F4Q8J6Lq0XgD4u6PoUPjO5htpAmlS2ZjFwXbG2EtCgGHwXYlfav04brTdgqYrlbYz8yfjZ8FfGEnw8/a90vRfBGvTx6rrug/2Ja2WlTv9thhaIM1soT96qBeSmQMHPevUNP8A2bdN+IX7Vnxe1nxz8OTrmlf8IjpdtpF5rGlvLbGb7NtlFuzrsMqlVGV+ZD0IzX3RikKjimI/NL4E/Crx3pdx+xUdQ8H+IrP/AIR5PES6ubrTJ4/7NEm8RC43J+53DG3fjOeK4u8/ZB+2/sPzXE3wkvpPiVdeNjLM50Wb+1zaG+KE/d80Q+Sc8fLj5uvNfrFgdutJtFDDax8ITfs63HhX9rH4iaT4B8Cr4a8G618JbjTIZ9P0w22ly6k02xEaUKIzLtwTyWIG45615j4Vm+Ivi7wn+zJ4El+C/j3Q3+Hni3TBrms6npZS0xF5kYeEglpItu5ml2hEwASd6k/p7sBoA5qlKzUuoraOJ+a/in9l2PxX4O/a/wBd1b4Y3OqeL7zXruTwze3GjSSXc0QCvG9iSm4guW+aL73IJI4rqvB37PI+F/7QP7J+p+EPh5NoFpF4bv4/FOoafpDoEnawXaL6UJxIZCwHmkNngdMV9/7fajaOOKUfdSQpLmv5nyR+2x4F8a2/xA+D/wAXvBfhqfxq3gLUrl9S8P2Ch7ye1uERHkgU/fdQhAUZbLqQMBiMH47+MfiH+13+zL8TNA8G/C7xr4IuPstr9nTxdbppt3qmJw9xbRQ7mIXyo8bywD79uPX7V29KbtA6DAqOXTlLvqn1Pzk+G/w9/wCFjftC/CvxH8Of2fvEPwU0nwhp98viS51nR00X+0zJbCOK2j2ndcEtuHnMNwDljjjPI/su/AXXf+Eq8a+Fbb4P6tbeCrnwpf2P234g+F7TT762vHZXS2W7jIXU4zKiMJpEyu1SuwD5v1K2j685o2ir6WF0sfI//BNrQf8AhD/gbF4c1D4Zax4A8U6SwttZvtX0RLD+15t8rJLHIPmuFRWC72HGcDIr64QYGAMAUBR6U4ALTZMY8uwtLSUtIoKKKKACiiigAooooAKKKKACiiigAooooAK+Vf8AgqP/AMmJ/E3/ALhn/p0tK+qq+Vf+Co3P7CvxMH/YM/8ATpaUAeVf8EVf+TWfFX/Y53X/AKQ2Nff9fAP/AARW4/ZZ8Vf9jndf+kNjX37QAtFJRQAtFJRQAtFJRQAtFJRQAtFJRQAtFJRQAtFJRQAUYoooAMUbaKKAEKg0bRjHP50tFAHP+Jvh54X8afZ/+Eh8O6Vrv2bd5H9p2Udx5W7G7bvU7c7VzjrtHpXkfir9h34R+Job7ytBm0O7upTL9q0u7kQwsX3MI43LRKDyu0JgAkKBxj3yiuuji8Rh3elUcfRsiUIy3R8Z+Kv+CbPhy4+zf8I74y1TTCu7zv7Uto7zfnG3Zs8nbj5s53ZyOmDnlNb/AOCamqwaZM+keOrO/wBRXHl297pr20TfMM7pFkkK4GSMIckAcA5H3swz9Kbt+XHQV68OIszhp7Zv1szL6vT7H5x/8O5fiX/0GPC34Xdx/wDI9eO/tOfA3Uv2UfBEXiDxj4j8Ny3F5IYNN0axup3vL+QY3eWrQgBEDKzuxCqCByzor/r/APrXhf7QH7FPwj/aY1R9Z8deHri78Qppn9lW2sWuo3FvNaRBpHRkRX8osjyuwMkbAkgMGAxXR/rNj7W93/wFC+rwPyY+CXwM0H9paMeKfiT8bvA/w08MiZxZeHb7XLZr6Xy22hpLdp4zErEuPMciRsEhAjI1fUOh/s4/ALSpbOBv2m/h+mlxSAyWtlfafATHuy6o32tgjEZ+ba2Cc4PSvHfBX7NPws/Zb+PEnwt/ad8FW+r+HPEExl8K/EyPUL6xtJF+VfLuFinVI1yVD5+aB3y7PDIkqfei/wDBLz9mQ8/8K0P/AIPtU/8AkmvMo5tjKEpTpTs5bvQ0dKD0ex554b+FP7H2i3rz33xk8M6/G0ZQW2oeNbBI1OQQ4MDRtkYI5YjDHjOCPVfCvir9kbwXDYppPi74TRS2Momt7ufXdPnuY5A5dX8+SRpCwY5BLZXAxjAxnf8ADrn9mI9fhn/5XtU/+SaT/h1z+zF/0TP/AMr2qf8AyTWFbMMZida1WT+Y1ThHZHqa/tWfA9enxj8Af+FRY/8Ax2l/4as+B/H/ABeTwB/4VFj/APHa8r/4dc/sxf8ARMv/ACvap/8AJNH/AA65/Zi/6Jl/5XtU/wDkmvPND1X/AIau+CA/5rJ4A/8ACosf/jtH/DV/wQ/6LJ4A/wDCosf/AI7XlX/Drn9mL/omX/le1T/5Jo/4dc/sxf8ARMv/ACvap/8AJNID1b/hrD4If9Fk+H//AIVFj/8AHaP+GsPgh/0WT4f/APhUWP8A8dryn/h1z+zF/wBEy/8AK9qn/wAk0f8ADrn9mL/omX/le1T/AOSaYHq3/DWHwQ/6LJ8P/wDwqLH/AOO0f8NYfBD/AKLJ8P8A/wAKix/+O15T/wAOuf2Yv+iZf+V7VP8A5Jo/4dc/sxf9Ey/8r2qf/JNAHq3/AA1h8EP+iyfD/wD8Kix/+O0f8NYfBD/osnw//wDCosf/AI7XlP8Aw65/Zi/6Jl/5XtU/+SaP+HXP7MX/AETL/wAr2qf/ACTQB6t/w1h8EP8Aosnw/wD/AAqLH/47R/w1h8EP+iyfD/8A8Kix/wDjteU/8Ouf2Yv+iZf+V7VP/kmj/h1z+zF/0TL/AMr2qf8AyTQB6t/w1h8EP+iyfD//AMKix/8AjtH/AA1h8EP+iyfD/wD8Kix/+O15T/w65/Zi/wCiZf8Ale1T/wCSaP8Ah1z+zF/0TL/yvap/8k0Ae0+Gf2hfhZ441610Tw58SvB/iDWbrf5GnaXr1rc3E21S7bY0kLNhVZjgcBSe1egAZrwH4W/sF/Ar4L+OtL8Y+DfAx0bxHpnm/ZLz+17+fy/MieJ/klnZDlJGHKnrkcgGvfxxQAm2l20UUAc98QPH3h74V+DtV8V+K9Vt9E8PaXCZ7y+uSdsa5AAAHLMzFVVFBZmZVUEkCsn4RfGjwX8efBsXirwH4gt/EWhPNJbG4hR42jlQgNHJHIqyRtgq211BKujD5WUk+NXwj8PfHj4Y6/4C8VRXEmh61CsUxtJjFNGyuskcqNz8ySIjgMGUlQGVlyp/Gb4hfCf49f8ABLX4jaX4m0TxB9t8Maje4i1CwMp0vVPLMoS11G2yAJTEzuEJbb5jmGUtGzIAfuevNLtr5/8A2U/21vh7+1hoKf8ACO3n9m+LrWyS61Xwtd5+0WeWKMUcqFniDAfvI+gki3rGzhK+gFOaADFG2looATFG2looATFIFxTqKAExS0UUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABXy9/wUa+EfxH+OH7Nt54S+GkVve6ndalaS6jps00ML31lGxYxxySgIrLMLeXl0ysTAMc7H+oaQqGoA/DfwX+xR+3D8N9Ll0zwhZeJ/C2myzG4ks9F8cWlnC8pVVMhSO8UFiqIN2M4UDtW/wD8M0/8FC/+gv4//wDDjQ//ACdX7UgbaWgD8Vf+Gaf+Chf/AEF/H/8A4caH/wCTqP8Ahmn/AIKF/wDQX8f/APhxof8A5Or9qqKAPxV/4Zp/4KF/9Bfx/wD+HGh/+TqP+Gaf+Chf/QX8f/8Ahxof/k6v2qooA/FX/hmn/goX/wBBfx//AOHGh/8Ak6j/AIZp/wCChf8A0F/H/wD4caH/AOTq/aqigD8Vf+Gaf+Chf/QX8f8A/hxof/k6j/hmn/goX/0F/H//AIcaH/5Or9qqKAPxV/4Zp/4KF/8AQX8f/wDhxof/AJOo/wCGaf8AgoX/ANBfx/8A+HGh/wDk6v2qooA/FX/hmn/goX/0F/H/AP4caH/5Oo/4Zp/4KF/9Bfx//wCHGh/+Tq/aqigD8Vf+Gaf+Chf/AEF/H/8A4caH/wCTqP8Ahmn/AIKF/wDQX8f/APhxof8A5Or9qqKAPxV/4Zp/4KF/9Bfx/wD+HGh/+TqP+Gaf+Chf/QX8f/8Ahxof/k6v2qooA/FX/hmn/goX/wBBfx//AOHGh/8Ak6j/AIZp/wCChf8A0F/H/wD4caH/AOTq/aqigD8Vf+Gaf+Chf/QX8f8A/hxof/k6j/hmn/goX/0F/H//AIcaH/5Or9qqKAPxV/4Zp/4KF/8AQX8f/wDhxof/AJOo/wCGaf8AgoX/ANBfx/8A+HGh/wDk6v2qooA/FX/hmn/goX/0F/H/AP4caH/5Oo/4Zp/4KF/9Bfx9/wCHGh/+Tq/aqigD8Vf+Gaf+Chf/AEF/H3/hxof/AJOo/wCGaf8AgoX/ANBbx9/4caH/AOTq/aqigD8NvGn7E/7cXxJ0qLTPF1j4n8VabDMLmOz1rxxaXkKShWUSBJLxgGCuw3AZwxHetXSf2U/2/dB0uz0zTLrxtp2m2UKW1rZ2nxBt4oYIkUKkaIt8AqqoACgYAAFftpRQB+Kv/DNP/BQv/oL+P/8Aw40P/wAnUf8ADNP/AAUL/wCgv4//APDjQ/8AydX7VUUAfir/AMM0/wDBQv8A6C/j/wD8OND/APJ1H/DNP/BQv/oL+P8A/wAOND/8nV+1VFAH4q/8M0/8FC/+gv4//wDDjQ//ACdR/wAM0/8ABQv/AKC/j/8A8OND/wDJ1ftVRQB+Kv8AwzT/AMFC/wDoL+P/APw40P8A8nUf8M0/8FC/+gv4/wD/AA40P/ydX7VUUAfir/wzT/wUL/6C/j//AMOND/8AJ1H/AAzT/wAFC/8AoL+P/wDw40P/AMnV+1VFAH4q/wDDNP8AwUL/AOgv4/8A/DjQ/wDydR/wzT/wUL/6C/j/AP8ADjQ//J1ftVRQB+Kv/DNP/BQv/oL+P/8Aw40P/wAnUf8ADNP/AAUL/wCgv4//APDjQ/8AydX7VUUAfir/AMM0/wDBQv8A6C/j/wD8OND/APJ1H/DNP/BQv/oL+P8A/wAOND/8nV+1VFAH4q/8M0/8FC/+gv4//wDDjQ//ACdR/wAM0/8ABQv/AKC/j/8A8OND/wDJ1ftVRQB+Kv8AwzT/AMFC/wDoL+P/APw40P8A8nUf8M0/8FC/+gv4/wD/AA40P/ydX7VUUAfisP2av+ChK5J1fx9/4cWH/wCTq81/aF+Ev7XHw/8Ahfe6h8YNe8Tr4KnmhtprbWvHMN9DdSlw0ca24u3MzBk8zaqMVEbPwEJH7JftVftLeHv2U/hLd+NtftrjUZGmWx0zS7UFWvr10d44jJgiJdsbszt0VGwHbajfkhp+i/Hr/gqx8ZNL1DV0/svwtYbrV9UtrKVNC0OJVjM4iDMfNuZN8beWXMj7kyyRIDGAeFfsneC/ih44+PHhm1+EE1xY+N7Wb7XDqkbbIdOiXCyz3DEMBAFfa6srCQP5YVzIEb+kReleafAX9nXwF+zj4MtvD3gfQrfTkWCOK81J40N9qTIWIkupgoMrZkkIz8qbyqKi4UelhQvSgBaKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKAGsT2o3HvXgvxr+PHiP4T/AB/+D3ho2emv4H8aXN1pd7fTRP8Aaba9Ee62VJPMCASMQMFCflbB7V578LP22NR8afthfEf4V6pp+n2HhHQre6OlasIZUnmntDEt2kjlyjBS8h+VVwFGeTSvfbz/AAHbufXm40bvzr5a+Ff7bGiw/s56V8V/jFq2j+C7LXdQuk0qC1gmdpLZJmjiAjVpZJX2oWZkUAAg7VHXU8cftcaDqngn4e+K/hr4z8J32i+I/FVpoMt1rVtfyeYJN2+3jjgjLw3JAUr56qgByxGRl9bCWp9INJt7UnmV4L4+/bo+CPwz1fUtN8ReNGtbjTbv+z7uW10i+u7eC6Ay0DTwwPH5oAOU3bhg5AxXl+rftuaH8JP2ovidonxL8dW+jeA7DStJn0OzksvMk86eMtKyCGJppM5UnO4LkdKm+tg6XPszdx70ua+Ufjd/wUS+G/wb1H4a+Xd2/iDRPGGLuXVrWWby7HTyxQ3QVIX80h1dTECrgqcivp3QNesvE+h6drGnTfaNO1C3ju7abay74pEDo2GAIyCOCARTW3N0Fc0qKQUtMYUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAcT8Xvgv4K+PXg2Xwr498P2/iLQ3mjuBbzO8bxyoflkjljZXjbBZdyMCVd1OVZgdXwD8P/AA78LvBuk+E/Cmk2+ieHtKhEFnY2wO2NckkknJZmYlmdiWZmZmJJJPQ0UAIFC9BiloooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKAPnf9uL4P8AiP4sfCPT5fBOnx6l468M65Y+INEt5J0g3zwSjK+Y7Kq5RnPLAZA74r5b8efsgfGXTfgJ8Ntb8J6BBefGO3u9em8QWa31rHgavFMs7+Y8giYx5iAAZucYJAzX6VMuTmm7Bgc9KlaMbd7I+Cf2nv2NfG1x4P8AgZJ8NU1a+uvhzYPpc2n+HvEMeh6lKrwxoZre9lRkjbdGd2V+ZXIHXjivDn7IPxIj+GvhCKLwJ4k0vXY/ixYeKdYt/E3jDT9auGtY4gJr37REkKsSxwUw0hK55zx+lbL83XNDLnNHe3UL7eR+MNv8btE0H4ieKdU8SaVfeK/AsXxIvNbXRdH8eafaxSXIuysM8eiyRC+kIO1sGRY2IyQADX3/AOG/gjr/APw1V8avG+o+HYW0HxF4asdN0m/uGgdpmEJWeHZuLoM7QdwCtgdcV7V/wo/4df8ACX/8JWPAXhj/AISrzTcf25/Y9v8AbvMPV/P2b93XnOea7Xb2p7pXF1uj87bP9mT4weEf2cf2bn0zwVDrXjX4ba9Nqmo+FX1i2tnmjeeVgFuSzRdCmTuJ+boSCK/QXQ7i8vNFsJ9Rsl03UJYI5LmyWUTC3lKgvHvAAfacjcAM4zV4L8uPyoVccU76WJURy0tJS0FBRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAH/9k=)

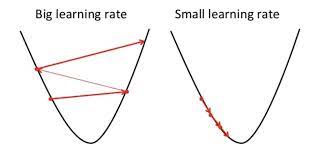
The intuition is that if you are hiking in a canyon and trying to descend most quickly down to the river at the bottom, you might look around yourself 360 degrees, find the direction where the ground is sloping the steepest, and walk downhill in that direction.

At first gradient descent takes a random value of our parameters from our function. Now we need an algorithm that will tell us whether at the next iteration we should move left or right to reach the minimum point. The gradient descent algorithm finds the slope of the loss function at that particular point and then in the next iteration, it moves in the opposite direction to reach the minima. Since we have a convex graph now we don’t need to worry about local minima. A convex curve will always have only 1 minima.

We can summarize the gradient descent algorithm as:

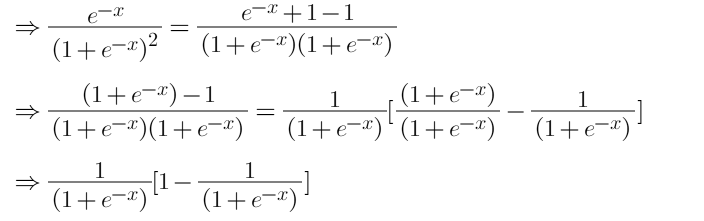
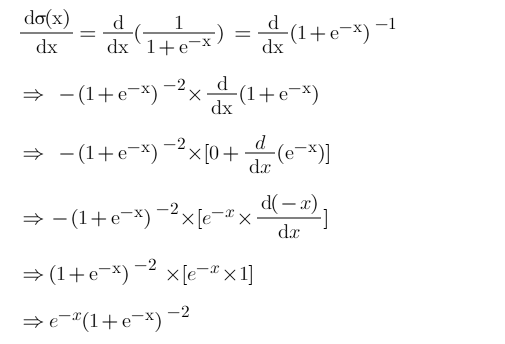


Here alpha is known as the learning rate. It determines the step size at each iteration while moving towards the minimum point. Usually, a lower value of “alpha” is preferred, because if the learning rate is a big number then we may miss the minimum point and keep on oscillating in the convex curve



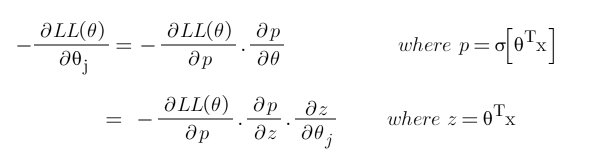
## Derivation of Cost Function:

Before we derive our cost function we’ll first find a derivative for our sigmoid function because it will be used in derivating the cost function.

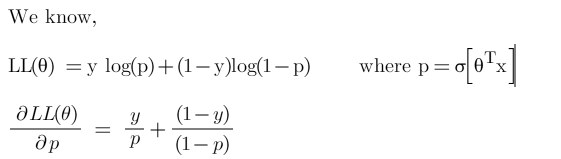


Now, we will derive the cost function with the help of the chain rule as it allows us to calculate complex partial derivatives by breaking them down.

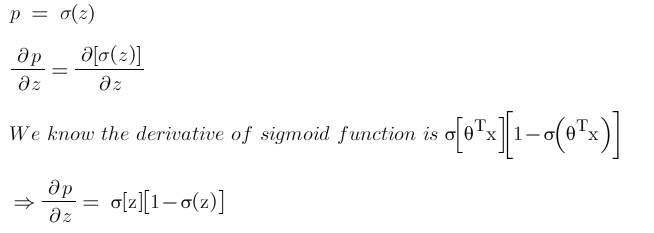
### Step-1: Use chain rule and break the partial derivative of log-likelihood.



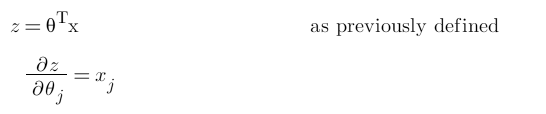
### Step-2: Find derivative of log-likelihood w.r.t p



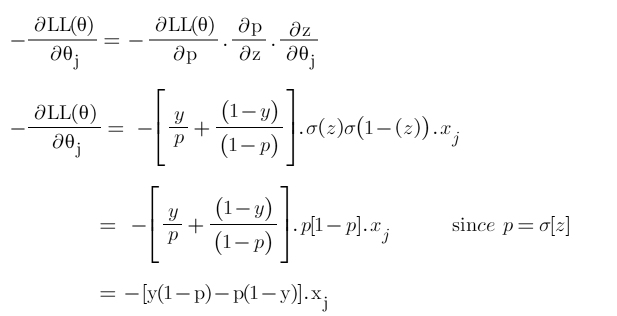
### Step-3: Find derivative of *****‘p’***** w.r.t *****‘z’*****

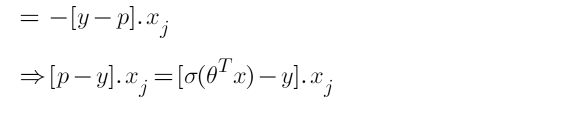


### *****Step-4: Find derivate of z w.r.t*****θ

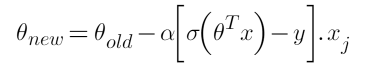


### Step-5: Put all the derivatives in equation 1





Hence the derivative of our cost function is:



Now since we have our derivative of the cost function, we can write our gradient descent algorithm as:

If the slope is negative (downward slope) then our gradient descent will add some value to our new value of the parameter directing it towards the minimum point of the convex curve. Whereas if the slope is positive (upward slope) the gradient descent will minus some value to direct it towards the minimum point.

Bernoulli Naive Bayes is a variant of the Naive Bayes algorithm, specifically designed for binary classification tasks where the features are binary-valued (e.g., presence or absence of a feature). It's based on the assumption that features are conditionally independent given the class label. Despite its simplifying assumptions, Naive Bayes often performs surprisingly well in practice and is computationally efficient. Here's a detailed explanation of Bernoulli Naive Bayes and its functionalities used in ML projects:

1. Model Representation:

- Bernoulli Naive Bayes models the probability of an instance belonging to a particular class given its binary-valued features.

- It uses the conditional probability of each feature given the class label and applies the Bayes' theorem to calculate the posterior probability of the class given the features.

3. Parameter Estimation:

- Bernoulli Naive Bayes estimates the parameters of the Bernoulli distribution for each feature given the class label from the training data.

- It calculates the probability of each feature being present or absent in instances belonging to each class.

4. Prediction:

- To classify a new instance, Bernoulli Naive Bayes computes the posterior probability of each class given the features using the Bayes' theorem.

- It assigns the instance to the class with the highest posterior probability.

5. Smoothing:

- Smoothing techniques such as Laplace smoothing or Lidstone smoothing can be applied to handle zero probabilities or alleviate the effects of overfitting, especially for rare features.

6. Evaluation:

- Bernoulli Naive Bayes models are evaluated using metrics such as accuracy, precision, recall, F1-score, and ROC AUC score, depending on the specific requirements of the classification task.

7. Implementation Example:

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import BernoulliNB

from sklearn.metrics import accuracy\_score, classification\_report

# Generate synthetic binary classification data

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train the Bernoulli Naive Bayes model

model = BernoulliNB()

model.fit(X\_train, y\_train)

# Make predictions on the testing set

y\_pred = model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Classification Report:")

print(report)

In this example:

- Synthetic binary classification data is generated using `make\_classification`.

- The data is split into training and testing sets using `train\_test\_split`.

- A Bernoulli Naive Bayes model is initialized and trained on the training data.

- Predictions are made on the testing set using `predict`.

- The model is evaluated using accuracy and a classification report generated using `classification\_report`.

This example demonstrates a basic implementation of Bernoulli Naive Bayes for binary classification using scikit-learn in Python.

Support Vector Machine (SVM) is a powerful supervised machine learning algorithm used for classification, regression, and outlier detection tasks. SVM aims to find the optimal hyperplane that best separates data points of different classes in the feature space. Here's a detailed explanation of SVM:

1. Objective:

- SVM's primary objective is to find the hyperplane with the maximum margin, which is the distance between the hyperplane and the nearest data points (support vectors) of each class.

- SVM seeks to achieve the best generalization performance by maximizing the margin, making it robust to overfitting.

2. Linear Separability:

- In the case of linearly separable data (i.e., data points of different classes can be perfectly separated by a hyperplane), SVM finds the optimal hyperplane that maximizes the margin between classes.

- The optimal hyperplane is determined by the support vectors, which are the data points closest to the hyperplane and influence its position.

3. Kernel Trick:

- SVM can handle non-linearly separable data by mapping the input features into a higher-dimensional space using kernel functions.

- Kernel functions (e.g., linear, polynomial, radial basis function) compute the dot product between input feature vectors in the transformed space, allowing SVM to find non-linear decision boundaries.

4. Margin Maximization:

- SVM finds the hyperplane that maximizes the margin while satisfying the following constraint: the distance between each data point and its corresponding class's hyperplane should be greater than or equal to a margin threshold (usually 1).

5. Regularization:

- SVM includes a regularization parameter (`C`) to control the trade-off between maximizing the margin and minimizing the classification error.

- A smaller value of `C` encourages a wider margin but may lead to more classification errors on the training data, while a larger value of `C` imposes a stricter penalty on misclassifications.

6. Dual Optimization Problem:

- SVM reformulates the optimization problem into its dual form, where the objective is to maximize the margin subject to constraints.

- The optimization problem is typically solved using optimization algorithms such as Sequential Minimal Optimization (SMO) or gradient descent.

7. Evaluation:

- SVM models are evaluated using metrics such as accuracy, precision, recall, F1-score, and ROC AUC score, depending on the specific requirements of the classification task.

- Cross-validation techniques are commonly used to assess the generalization performance of SVM models on unseen data.

8. Kernel Selection and Hyperparameter Tuning:

- The choice of kernel function and its parameters significantly impact the performance of SVM.

- Hyperparameter tuning is crucial to optimize model performance and generalization.

SVM is widely used in various domains, including text classification, image recognition, bioinformatics, and finance, due to its effectiveness in handling high-dimensional data and non-linear decision boundaries. It is a versatile algorithm with robust theoretical foundations and has been successfully applied to a wide range of machine learning tasks.

7. Implementation Example:

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, classification\_report

# Generate synthetic binary classification data

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train the SVM model

model = SVC(kernel='linear', C=1.0)

model.fit(X\_train, y\_train)

# Make predictions on the testing set

y\_pred = model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Classification Report:")

print(report)

In this example:

- Synthetic binary classification data is generated using `make\_classification`.

- The data is split into training and testing sets using `train\_test\_split`.

- An SVM model with a linear kernel is initialized and trained on the training data.

- Predictions are made on the testing set using `predict`.

- The model is evaluated using accuracy and a classification report generated using `classification\_report`.

This example demonstrates a basic implementation of SVM for binary classification using scikit-learn in Python.

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic used in natural language processing (NLP) and information retrieval to evaluate the importance of a word in a document relative to a collection of documents. It quantifies the importance of a term within a document in a corpus by considering both the frequency of the term in the document (TF) and the inverse document frequency (IDF) of the term across the entire corpus. Here's a detailed explanation of TF-IDF and its functionalities used in ML projects:

1. Term Frequency (TF):

- Term Frequency measures the frequency of a term (word) within a document relative to the total number of words in that document.

- It is calculated as the number of times a term appears in a document divided by the total number of terms in the document.

- TF is designed to capture the importance of a term within a specific document.

2. Inverse Document Frequency (IDF):

- Inverse Document Frequency measures the specificity or rarity of a term across the entire corpus.

- It is calculated as the logarithm of the ratio of the total number of documents in the corpus to the number of documents containing the term

- IDF penalizes common terms that appear in many documents and gives higher weight to rare terms that are more discriminative.

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic that reflects the importance of a word in a document relative to a collection of documents. It is commonly used in natural language processing (NLP) and information retrieval for text mining and document classification tasks. Here's how TF-IDF is calculated:

1. Term Frequency (TF):

- Term Frequency measures the frequency of a term (word) in a document.

- It is calculated by dividing the number of occurrences of a term in a document by the total number of terms in the document.

- The TF formula is given by:

```

TF(t, d) = (Number of times term t appears in document d) / (Total number of terms in document d)

```

2. Inverse Document Frequency (IDF):

- Inverse Document Frequency measures the importance of a term across a collection of documents.

- It is calculated as the logarithm of the ratio of the total number of documents to the number of documents containing the term, with an added smoothing term to prevent division by zero.

- The IDF formula is given by:

```

IDF(t, D) = log\_e (Total number of documents in collection D / Number of documents containing term t)

```

3. TF-IDF Calculation:

- TF-IDF is calculated by multiplying the Term Frequency (TF) of a term in a document by its Inverse Document Frequency (IDF) across the entire collection of documents.

- The TF-IDF formula is given by:

```

TF-IDF(t, d, D) = TF(t, d) IDF(t, D)

```

- This gives a higher weight to terms that are common in a document but rare across the entire document collection, indicating their importance in distinguishing documents.

4. Example:

- Let's consider a collection of three documents and calculate TF-IDF for a specific term 'apple':

- Document 1: "I like to eat apples"

- Document 2: "Apples are delicious"

- Document 3: "I prefer bananas over apples"

- Term Frequency (TF) for 'apple' in each document:

- TF(apple, Document 1) = 1/5 = 0.2

- TF(apple, Document 2) = 1/3 ≈ 0.33

- TF(apple, Document 3) = 1/5 = 0.2

- Total number of documents in the collection (D): 3

- Number of documents containing the term 'apple': 2 (Document 1 and Document 2)

- Inverse Document Frequency (IDF) for 'apple':

- IDF(apple, D) = log(3 / 2) ≈ 0.18

- TF-IDF for 'apple' in each document:

- TF-IDF(apple, Document 1) ≈ 0.2 0.18 ≈ 0.036

- TF-IDF(apple, Document 2) ≈ 0.33 0.18 ≈ 0.059

- TF-IDF(apple, Document 3) ≈ 0.2 0.18 ≈ 0.036

TF-IDF assigns higher weights to terms that are frequent in a document but rare in other documents, effectively capturing the importance of terms in the context of a document collection. It is commonly used in various NLP tasks such as text classification, information retrieval, and document clustering.

4. Functionalities:

- TF-IDF is widely used in ML projects for various NLP tasks, including document classification, information retrieval, text summarization, and clustering.

- It helps in identifying and weighting important terms in documents, allowing ML models to focus on discriminative features and improve performance.

- TF-IDF can be used to convert raw text data into numerical feature vectors suitable for input to ML algorithms such as SVM, logistic regression, or neural networks.

5. Implementation Example:

from sklearn.feature\_extraction.text import TfidfVectorizer

# Sample corpus of documents

corpus = [

"This is the first document.",

"This document is the second document.",

"And this is the third one.",

"Is this the first document?",

]

# Initialize TF-IDF vectorizer

tfidf\_vectorizer = TfidfVectorizer()

# Fit and transform the corpus into TF-IDF vectors

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(corpus)

# Get the feature names (terms)

feature\_names = tfidf\_vectorizer.get\_feature\_names\_out()

# Display TF-IDF matrix

print("TF-IDF Matrix:")

print(tfidf\_matrix.toarray())

# Display feature names

print("Feature Names:")

print(feature\_names)

In this example:

- We have a sample corpus of documents.

- We initialize a TF-IDF vectorizer using `TfidfVectorizer` from scikit-learn.

- We fit and transform the corpus into TF-IDF vectors using `fit\_transform`.

- We retrieve the feature names (terms) using `get\_feature\_names\_out`.

- Finally, we display the TF-IDF matrix and feature names.

This example demonstrates how to compute TF-IDF representation of documents using scikit-learn in Python.

Certainly! Accuracy and F1 Score are both evaluation metrics commonly used in machine learning projects, especially for classification tasks. Here's a detailed explanation of each metric along with its functionalities and a sample implementation:

1. Accuracy:

- Accuracy is a straightforward metric that measures the proportion of correctly classified instances out of the total number of instances in the dataset.

- It is calculated as the ratio of the number of correct predictions to the total number of predictions made by the model.

- Accuracy is suitable for balanced datasets where the number of instances in each class is approximately equal.

- However, accuracy can be misleading in the presence of class imbalance, where one class dominates the dataset, as the model can achieve high accuracy by simply predicting the majority class.

- Despite its limitations, accuracy provides a general indication of the overall correctness of the model's predictions.

2. F1 Score:

- F1 Score is a harmonic mean of precision and recall, providing a balanced measure of a model's performance that considers both false positives and false negatives.

- Precision measures the proportion of true positive predictions out of all positive predictions made by the model, while recall measures the proportion of true positive predictions out of all actual positive instances in the dataset.

- F1 Score ranges from 0 to 1, where a higher value indicates better model performance in terms of both precision and recall.

- F1 Score is particularly useful when dealing with imbalanced datasets, where precision and recall provide complementary information about the model's performance on different classes.

3. Functionality:

- Accuracy and F1 Score are both used to evaluate the performance of classification models and compare different models based on their predictive accuracy and balance between precision and recall.

- ML practitioners typically compute accuracy and F1 Score on a hold-out test set or using cross-validation techniques to assess the generalization performance of the model on unseen data.

- Both metrics can be easily calculated using built-in functions provided by libraries such as scikit-learn in Python.

4. Implementation Example:

- Below is an example of calculating accuracy and F1 Score for a binary classification model using scikit-learn in Python:

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, f1\_score

# Generate synthetic binary classification data

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train a logistic regression model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Make predictions on the testing set

y\_pred = model.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

# Calculate F1 Score

f1 = f1\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("F1 Score:", f1)

In this example:

- Synthetic binary classification data is generated using `make\_classification`.

- The data is split into training and testing sets using `train\_test\_split`.

- A logistic regression model is initialized and trained on the training data.

- Predictions are made on the testing set using `predict`.

- Accuracy and F1 Score are calculated using `accuracy\_score` and `f1\_score` functions from scikit-learn.

This example demonstrates how to calculate accuracy and F1 Score for a binary classification model using scikit-learn in Python. Both metrics provide valuable insights into the performance of the model, allowing ML practitioners to evaluate and compare different models effectively.

Binary Classification Problem:

Let's consider a binary classification problem where we want to predict whether an email is spam (positive class) or not spam (negative class) based on some features.

Example Data:

Suppose we have the following data:

- True labels (ground truth): [0, 1, 1, 0, 1, 0, 0, 1, 1, 1]

- Predicted labels: [0, 1, 1, 0, 1, 1, 0, 1, 0, 1]

Step-by-Step Calculation:

1. Accuracy Calculation:

Accuracy measures the proportion of correctly predicted instances among all instances.

true\_labels = [0, 1, 1, 0, 1, 0, 0, 1, 1, 1]

predicted\_labels = [0, 1, 1, 0, 1, 1, 0, 1, 0, 1]

# Calculate accuracy

correct\_predictions = sum(1 for true, pred in zip(true\_labels, predicted\_labels) if true == pred)

accuracy = correct\_predictions / len(true\_labels)

print("Accuracy:", accuracy)

2. F1 Score Calculation:

F1 score is the harmonic mean of precision and recall. Precision measures the proportion of true positive predictions among all positive predictions, while recall measures the proportion of true positive predictions among all actual positive instances.

true\_positives = sum(1 for true, pred in zip(true\_labels, predicted\_labels) if true == pred == 1)

false\_positives = sum(1 for true, pred in zip(true\_labels, predicted\_labels) if true == 0 and pred == 1)

false\_negatives = sum(1 for true, pred in zip(true\_labels, predicted\_labels) if true == 1 and pred == 0)

# Calculate precision

precision = true\_positives / (true\_positives + false\_positives) if (true\_positives + false\_positives) > 0 else 0

# Calculate recall

recall = true\_positives / (true\_positives + false\_negatives) if (true\_positives + false\_negatives) > 0 else 0

# Calculate F1 score

f1\_score = 2 (precision recall) / (precision + recall) if (precision + recall) > 0 else 0

print("Precision:", precision)

print("Recall:", recall)

print("F1 Score:", f1\_score)

### Results:

- Accuracy: 0.8

- Precision: 0.8333

- Recall: 0.8333

- F1 Score: 0.8333

This example illustrates how to calculate accuracy and F1 score step by step for a binary classification problem. These metrics are essential for evaluating the performance of classification models in machine learning projects.

**Coding Module**

**How to Do Twitter Sentiment Analysis?**

In this module we are going to analyze Twitter sentiment analysis using machine learning algorithms, the sentiment of tweets provided from the **Sentiment140 dataset**by developing a machine learning pipeline involving the use of three classifiers (**Logistic Regression, Bernoulli Naive Bayes, and SVM**) along with using **Term Frequency- Inverse Document Frequency**(**TF-IDF)**. The performance of these classifiers is then evaluated using **accuracy** and **F1 Scores**.

For data preprocessing, we will be using Natural Language Processing’s (NLP) NLTK library.

Twitter Sentiment Analysis: Problem Statement

In this project, we try to implement an NLP **Twitter sentiment analysis model** that helps to overcome the challenges of sentiment classification of tweets. We will be classifying the tweets into positive or negative sentiments. The necessary details regarding the dataset involving the Twitter sentiment analysis project are:

The dataset provided is the **Sentiment140 Dataset**which consists of **1,600,000 tweets** that have been extracted using the Twitter API. The various columns present in this Twitter data are:

* **target:**the polarity of the tweet (positive or negative)
* **ids:**Unique id of the tweet
* **date:**the date of the tweet
* **flag:**It refers to the query. If no such query exists, then it is NO QUERY.
* **user:** It refers to the name of the user that tweeted
* **text:** It refers to the text of the tweet

**Details of Dataset:**

The Sentiment140 dataset is a widely used dataset in natural language processing (NLP) and sentiment analysis tasks. It was created by Stanford University researchers for the purpose of sentiment analysis and consists of tweets labelled with sentiment polarity.

**Data Source:** The dataset is compiled from Twitter, a popular social media platform where users post short messages called tweets. The tweets are publicly available and cover various topics and domains.

**Labelling**: Each tweet in the dataset is labelled with sentiment polarity, indicating whether the sentiment expressed in the tweet is positive, negative, or neutral. The sentiment polarity is determined based on emoticons (e.g., ":)" for positive sentiment, ":(" for negative sentiment) present in the tweets.

**Size**: The dataset originally contained 1.6 million tweets. However, subsets of this dataset are often used in research and experimentation due to its large size.

**Data Format**: The dataset is typically provided in a CSV (Comma Separated Values) format, where each row represents a tweet and contains the following columns:

- Polarity: Indicates the sentiment polarity of the tweet (0 for negative, 2 for neutral, 4 for positive).

- ID: Unique identifier for the tweet.

- Date: Date and time when the tweet was posted.

- Query: Indication of the query used to retrieve the tweet (often empty or irrelevant).

- User: Twitter username of the user who posted the tweet.

- Text: The actual text content of the tweet.

**Preprocessing**: The dataset often requires preprocessing steps, such as removing URLs, mentions, special characters, and stopwords, as well as tokenization and lowercasing, before being used for sentiment analysis tasks.

**Applications**: The Sentiment140 dataset has been widely used for training and evaluating sentiment analysis models, sentiment classification algorithms, and other NLP tasks. Researchers and practitioners use this dataset to develop and benchmark their sentiment analysis techniques.

**Challenges**: While the dataset provides a large and diverse collection of tweets, it also has some limitations. The use of emoticons for sentiment labelling may not always accurately reflect the sentiment expressed in the text. Additionally, tweets are often noisy and informal, which can pose challenges for sentiment analysis algorithms.

Overall, the Sentiment140 dataset has been instrumental in advancing research in sentiment analysis and remains a benchmark dataset in the field of natural language processing.

**Twitter Sentiment Analysis: Project Pipeline**

The various steps involved in the **Machine Learning Pipeline** are:

* Import Necessary Dependencies
* Read and Load the Dataset
* Exploratory Data Analysis
* Data Visualization of Target Variables
* Data Preprocessing
* Splitting our data into Train and Test sets.
* Transforming Dataset using TF-IDF Vectorizer
* Function for Model Evaluation
* Model Building
* Model Evaluation

**Step-1: Import the Necessary Dependencies**

# utilities

import re

import numpy as np

import pandas as pd

# plotting

import seaborn as sns

from wordcloud import WordCloud

import matplotlib.pyplot as plt

# nltk

from nltk.stem import WordNetLemmatizer

# sklearn

from sklearn.svm import LinearSVC

from sklearn.naive\_bayes import BernoulliNB

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics import confusion\_matrix, classification\_report

**Code Explanation:**

**Importing Libraries:**

* re: Regular Expression library for pattern matching and manipulation of strings.
* numpy and pandas: Libraries for numerical computing and data manipulation, respectively.
* seaborn, matplotlib.pyplot, and wordcloud: Libraries for data visualization and creating word clouds.
* nltk: Natural Language Toolkit library for natural language processing tasks.
* sklearn: Scikit-learn library, which provides tools for machine learning tasks.
* LinearSVC, BernoulliNB, and LogisticRegression: Classes for Support Vector Classification, Bernoulli Naive Bayes, and Logistic Regression classifiers, respectively.
* train\_test\_split: Function to split data into training and testing sets.
* TfidfVectorizer: Class for converting a collection of raw documents to a matrix of TF-IDF features.
* confusion\_matrix and classification\_report: Functions to evaluate the performance of a classification model.

This code snippet imports various libraries and tools necessary for data preprocessing, visualization, natural language processing, and building machine learning models for sentiment analysis or text classification tasks.

**Step-2: Read and Load the Dataset**

# Importing the dataset

DATASET\_COLUMNS = ['target', 'ids', 'date', 'flag', 'user', 'text']

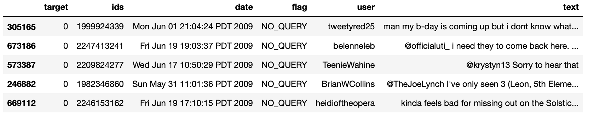
DATASET\_ENCODING = "ISO-8859-1"

file\_path = r'C:\Users\hp\OneDrive\Desktop\Sentimental\_Analysis\_Project\training.1600000.processed.noemoticon.csv'

df = pd.read\_csv(file\_path, encoding=DATASET\_ENCODING, names=DATASET\_COLUMNS)

df.sample(5)

**Output:**



**Code Explanation:**

This code segment imports a dataset into a pandas DataFrame (`df`). Here's what each part of the code does:

DATASET\_COLUMNS=['target','ids','date','flag','user','text']

This line defines the column names for the dataset. It seems like the dataset has six columns: 'target', 'ids', 'date', 'flag', 'user', and 'text'.

DATASET\_ENCODING = "ISO-8859-1"

This line specifies the encoding of the dataset file. In this case, the encoding is set to "ISO-8859-1".

df = pd.read\_csv('C:/Talent Battle/4 Weeks ML Project Challenge/Dataset/training.1600000.processed.noemoticon.csv', encoding=DATASET\_ENCODING, names=DATASET\_COLUMNS)

This line reads the dataset file located at the specified path (`'C:/Talent Battle/4 Weeks ML Project Challenge/Dataset/training.1600000.processed.noemoticon.csv'`) into a pandas DataFrame (`df`). It uses the `pd.read\_csv()` function from the pandas library to read the CSV file. The `encoding` parameter specifies the encoding used to interpret the file. The `names` parameter assigns the column names defined earlier (`DATASET\_COLUMNS`) to the DataFrame.

df.sample(5)

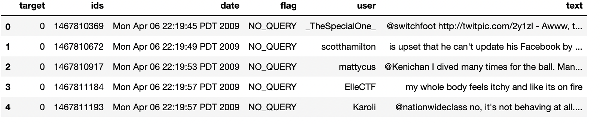
This line displays a random sample of 5 rows from the DataFrame `df`, allowing you to inspect the imported data. It helps to verify that the dataset has been imported correctly and to get a glimpse of its structure.

**Step-3: Exploratory Data Analysis**

**3.1: Five top records of data**

df.head()

**Output:**



**Code Explanation:**

The `df.head()` function displays the first few rows of the DataFrame `df`. This is a commonly used method in pandas to quickly inspect the structure and contents of a DataFrame. By default, it displays the first 5 rows, but you can specify the number of rows you want to see by passing an integer argument to the function.

df.head()

The output show the first 5 rows of the DataFrame `df`, including the column names and the data contained within each cell. This is useful for understanding the dataset's structure, the types of data it contains, and any potential issues with the data import process.

**3.2: Columns/features in data**

df.columns

**Output:**

Index(['target', 'ids', 'date', 'flag', 'user', 'text'], dtype='object')

**Code Explanation:**

The `df.columns` attribute returns an Index object containing the column labels of the DataFrame `df`. It essentially provides a list-like view of the column names in the DataFrame.

The output would display the column labels (names) of the DataFrame `df`. Each column label represents a variable or feature in the dataset. It's a convenient way to quickly inspect the column names and understand the structure of the DataFrame.

**3.3: Length of the dataset**

print('length of data is', len(df))

**Output:**

length of data is 1600000

**Code Explanation:**

This code snippet prints the length of the DataFrame `df`, which corresponds to the number of rows in the DataFrame. Here's what each part of the code does:

print('length of data is', len(df))

- `print()`: This function is used to display output to the console.

- `'length of data is'`: This is a string literal that serves as part of the message to be displayed.

- `,`: This comma separates the different items that are passed to the `print()` function.

- `len(df)`: This calculates the length of the DataFrame `df`, which corresponds to the number of rows in the DataFrame. The `len()` function returns the number of elements in the DataFrame's index, which in this case is the number of rows.

When you execute this code, it will print a message to the console indicating the length (number of rows) of the DataFrame.

**3.4: Shape of data**

df.shape

**Output:**

(1600000, 6)

**Code Explanation:**

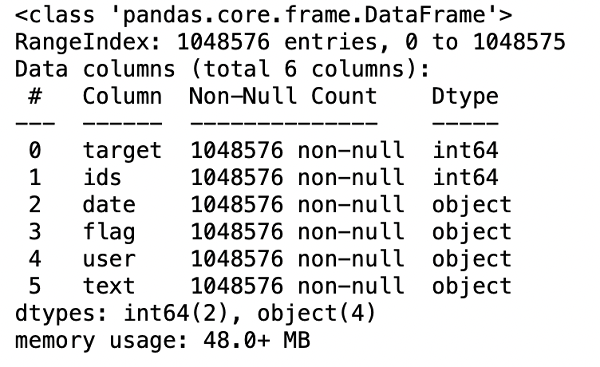
The `df.shape` attribute returns a tuple representing the dimensions of the DataFrame `df`. The tuple contains two elements: the number of rows and the number of columns, respectively.

The output would be a tuple with two values: the first value represents the number of rows in the DataFrame, and the second value represents the number of columns.

**3.5: Data information**

df.info()

**Output:**



**Code Explanation:**

The df.info() function provides a concise summary of the DataFrame df, including information about the index, data types, memory usage, and non-null values.

Here's what each part of the output represents:

Index: Information about the index, including the number of entries and data type.

Columns: Information about each column, including the column name, non-null count, and data type.

Dtype: Data type of each column.

Memory Usage: Total memory usage of the DataFrame.

This function is helpful for quickly understanding the structure and composition of the DataFrame, such as identifying missing values and data types. It provides a summary that is especially useful when dealing with large datasets.

**3.6: Datatypes of all columns**

df.dtypes

**Output:**

target int64

ids int64

date object

flag object

user object

text object

dtype: object

**Code Explanation:**

The `df.dtypes` attribute returns a Series containing the data types of each column in the DataFrame `df`. The index of the Series contains the column names, and the values contain the corresponding data types.

Here's how you would use it:

df.dtypes

The output would be a Series where each row represents a column in the DataFrame, and the values indicate the data type of each column. For example:

target int64

ids int64

date object

flag object

user object

text object

dtype: object

In this example, the 'target' and 'ids' columns have data type `int64`, while the 'date', 'flag', 'user', and 'text' columns have data type `object`. The `object` data type typically indicates strings or mixed types.

**3.7: Checking for null values**

np.sum(df.isnull().any(axis=1))

**Output**

0

**Code Explanation:**

This code snippet calculates the number of rows in the DataFrame `df` that contain at least one null value. Here's what each part of the code does:

np.sum(df.isnull().any(axis=1))

- `df.isnull()`: This DataFrame method returns a DataFrame of the same shape as `df` where each element is `True` if it's a null value and `False` otherwise.

- `.any(axis=1)`: This method checks if any value along the specified axis (axis=1, which corresponds to rows) is `True`. It returns a Series where each row indicates whether there is at least one null value in that row.

- `np.sum()`: This NumPy function calculates the sum of the values in the Series. Since `True` is treated as 1 and `False` as 0, summing up the values effectively counts the number of rows where at least one null value is present.

So, `np.sum(df.isnull().any(axis=1))` calculates the total number of rows in the DataFrame `df` that contain at least one null value.

**3.8: Rows and columns in the dataset**

print('Count of columns in the data is: ', len(df.columns))

print('Count of rows in the data is: ', len(df))

**Output:**

Count of columns in the data is: 6

Count of rows in the data is: 1600000

**Code Explanation:**

This code snippet prints the count of columns and rows in the DataFrame `df`. Here's what each part of the code does:

print('Count of columns in the data is: ', len(df.columns))

- `len(df.columns)`: This calculates the number of columns in the DataFrame `df` by taking the length of the `columns` attribute, which returns an Index object containing the column labels.

print('Count of rows in the data is: ', len(df))

- `len(df)`: This calculates the number of rows in the DataFrame `df` by taking the length of the DataFrame itself. The `len()` function returns the number of elements in the DataFrame's index, which in this case is the number of rows.

Together, these two lines print the count of columns and rows in the DataFrame `df`.

When executed, the output would look something like this:

Count of columns in the data is: 6

Count of rows in the data is: 1600000

This indicates that the DataFrame has 6 columns and 1,600,000 rows.

**3.9: Check unique target values**

df['target'].unique()

**Output:**

array([0, 4], dtype=int64)

**Code Explanation:**

The code `df['target'].unique()` returns an array containing the unique values present in the 'target' column of the DataFrame `df`. Here's what each part of the code does:

- `df['target']`: This accesses the 'target' column of the DataFrame `df`.

- `.unique()`: This method returns an array containing the unique values present in the specified column.

For example, if the 'target' column contains values like `[0, 4]`, running `df['target'].unique()` would return an array `array([0, 4])`, indicating that there are two unique values in the 'target' column: 0 and 4.

This code is useful for understanding the unique categories or labels present in a categorical column, which can be important for various data analysis and modeling tasks.

**3.10: Check the number of target values**

df['target'].nunique()

**Output:**

2

**Code Explanation:**

The code `df['target'].nunique()` returns the number of unique values present in the 'target' column of the DataFrame `df`. Here's what each part of the code does:

- `df['target']`: This accesses the 'target' column of the DataFrame `df`.

- `.nunique()`: This method calculates the number of unique values present in the specified column.

For example, if the 'target' column contains values like `[0, 4, 0, 4, 0]`, running `df['target'].nunique()` would return `2`, indicating that there are two unique values in the 'target' column: 0 and 4.

This code is useful for quickly determining the number of distinct categories or labels present in a categorical column, which can provide insights into the diversity of the data and inform subsequent data analysis or modeling decisions.

Certainly! The difference between `df['target'].unique()` and `df['target'].nunique()` lies in what they return:

1. `df['target'].unique()`: This statement returns an array containing all the unique values present in the 'target' column of the DataFrame `df`. Each unique value is listed only once in the array. This is useful when you want to see all the distinct values present in a column.

2. `df['target'].nunique()`: This statement returns a single integer value representing the count of unique values in the 'target' column of the DataFrame `df`. It provides a numeric summary of the diversity of values in the column without listing the actual unique values themselves.

In summary:

- Use `df['target'].unique()` when you want to see the actual unique values present in a column.

- Use `df['target'].nunique()` when you want to quickly determine the count of unique values in a column without listing the values themselves.

**Step-4: Data Visualization of Target Variables**

# Plotting the distribution for dataset.

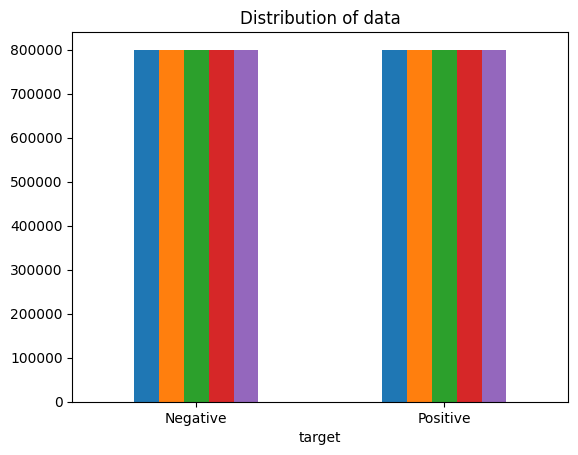
ax = df.groupby('target').count().plot(kind='bar', title='Distribution of data',legend=False)

ax.set\_xticklabels(['Negative','Positive'], rotation=0)

# Storing data in lists.

text, sentiment = list(df['text']), list(df['target'])

**Output:**



**Code Explanation:**

This code snippet plots the distribution of data in the DataFrame `df` based on the 'target' column, which likely represents sentiment labels (e.g., positive or negative). Here's a breakdown of what each part of the code does:

ax = df.groupby('target').count().plot(kind='bar', title='Distribution of data', legend=False)

- `df.groupby('target').count()`: This groups the DataFrame `df` by the values in the 'target' column and counts the occurrences of each value. This results in a new DataFrame with the count of occurrences for each unique value in the 'target' column.

- `.plot(kind='bar', title='Distribution of data', legend=False)`: This creates a bar plot using the counts obtained from the `groupby` operation. The `kind='bar'` parameter specifies the type of plot to create, and `title='Distribution of data'` sets the title of the plot. The `legend=False` parameter removes the legend from the plot.

ax.set\_xticklabels(['Negative','Positive'], rotation=0)

- `ax.set\_xticklabels(['Negative','Positive'], rotation=0)`: This sets the labels for the x-axis ticks of the plot. The first argument `['Negative','Positive']` specifies the labels to be displayed, and `rotation=0` ensures that the labels are not rotated.

text, sentiment = list(df['text']), list(df['target'])

- `text`: This variable stores the values from the 'text' column of the DataFrame `df`. It likely contains the text content of the dataset.

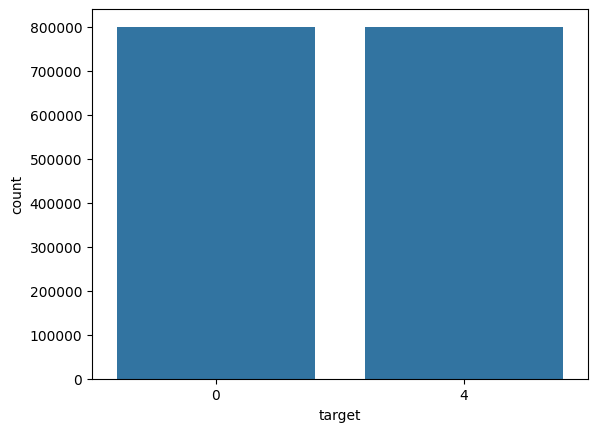
- `sentiment`: This variable stores the values from the 'target' column of the DataFrame `df`. It likely contains the sentiment labels associated with each text.

Overall, this code snippet generates a bar plot showing the distribution of sentiment labels in the dataset, with 'Negative' and 'Positive' labels on the x-axis and the corresponding counts on the y-axis.

import seaborn as sns

sns.countplot(x='target', data=df)

**Output:**



**Code Explanation:**

This code snippet utilizes the seaborn library to create a count plot showing the distribution of values in the 'target' column of the DataFrame `df`. Here's a breakdown of what each part of the code does:

import seaborn as sns

This line imports the seaborn library, commonly used for statistical data visualization in Python.

sns.countplot(x='target', data=df)

This line creates a count plot using seaborn's `countplot()` function. The `x` parameter specifies the column to plot on the x-axis, which in this case is 'target'. The `data` parameter specifies the DataFrame containing the data to be plotted, which is `df` in this case.

The count plot visually represents the distribution of values in the 'target' column, with each unique value being represented by a bar whose height corresponds to the frequency of that value in the dataset.

Overall, this code snippet provides a concise way to visualize the distribution of sentiment labels in the dataset using seaborn's count plot functionality.

**Step-5: Data Preprocessing**

In the above-given problem statement, before training the model, we performed various pre-processing steps on the dataset that mainly dealt with removing stopwords, removing special characters like emojis, hashtags, etc. The text document is then converted into lowercase for better generalization.

Subsequently, the punctuations were cleaned and removed, thereby reducing the unnecessary noise from the dataset. After that, we also removed the repeating characters from the words along with removing the URLs as they do not have any significant importance.

At last, we then performed Stemming (reducing the words to their derived stems) and Lemmatization (reducing the derived words to their root form, known as lemma) for better results.

**5.1: Selecting the text and Target column for our further analysis**

data=df[['text','target']]

**Code Explanation:**

This line of code selects specific columns ('text' and 'target') from the DataFrame `df` and creates a new DataFrame called `data` containing only these columns. Here's what each part of the code does:

data = df[['text', 'target']]

- `df[['text', 'target']]`: This part of the code uses double square brackets to access specific columns ('text' and 'target') from the original DataFrame `df`. When you use double square brackets, you are selecting multiple columns and creating a new DataFrame with those columns.

- `data = ...`: This assigns the selected columns to a new DataFrame called `data`. Now, `data` contains only the 'text' and 'target' columns from the original DataFrame `df`.

In summary, this line of code creates a new DataFrame `data` that contains only the 'text' and 'target' columns from the original DataFrame `df`, allowing you to work with a subset of the original data. This can be useful for various data analysis and modeling tasks, especially when you only need specific columns of the dataset.

**5.2: Replacing the values to ease understanding. (Assigning 1 to Positive sentiment 4)**

**5.3: Printing unique values of target variables**

data['target'].unique()

**Output:**

array([0, 1], dtype=int64)

**Code Explanation:**

This line of code retrieves the unique values present in the 'target' column of the DataFrame `data`. Here's the breakdown:

data['target'].unique()

- `data['target']`: This part of the code accesses the 'target' column of the DataFrame `data`.

- `.unique()`: This method is called on the 'target' column, which returns an array containing the unique values present in that column.

So, when you execute `data['target'].unique()`, it will return an array containing all the unique values present in the 'target' column of the DataFrame `data`. This allows you to see the distinct categories or labels present in the 'target' column.

**5.4: Separating positive and negative tweets**

data\_pos = data[data['target'] == 1]

data\_neg = data[data['target'] == 0]

**Code Explanation:**

This code snippet creates two separate DataFrames: `data\_pos` and `data\_neg`. Here's what each part of the code does:

data\_pos = data[data['target'] == 1]

This line of code filters the DataFrame `data` to include only rows where the value in the 'target' column is equal to 1. This effectively creates a new DataFrame called `data\_pos` containing only the rows where the sentiment label is positive (assuming 1 represents positive sentiment).

data\_neg = data[data['target'] == 0]

Similarly, this line of code filters the DataFrame `data` to include only rows where the value in the 'target' column is equal to 0. This creates a new DataFrame called `data\_neg` containing only the rows where the sentiment label is negative (assuming 0 represents negative sentiment).

After executing these two lines of code, you have two separate DataFrames: `data\_pos` containing rows with positive sentiment and `data\_neg` containing rows with negative sentiment. These DataFrames can be used for further analysis or modeling tasks specific to each sentiment category.

**5.5: Taking one-fourth of the data so we can run it on our machine easily**

data\_pos = data\_pos.iloc[:int(20000)]

data\_neg = data\_neg.iloc[:int(20000)]

**Code Explanation:**

This code snippet selects the first 20,000 rows from each of the `data\_pos` and `data\_neg` DataFrames and updates them with the selected subset. Here's the breakdown:

data\_pos = data\_pos.iloc[:int(20000)]

This line of code selects the first 20,000 rows from the DataFrame `data\_pos` using integer-based indexing with `iloc`. It slices the DataFrame up to the 20,000th row (exclusive) and updates `data\_pos` with this subset.

data\_neg = data\_neg.iloc[:int(20000)]

Similarly, this line of code selects the first 20,000 rows from the DataFrame `data\_neg` using integer-based indexing with `iloc`. It slices the DataFrame up to the 20,000th row (exclusive) and updates `data\_neg` with this subset.

After executing these two lines of code, both `data\_pos` and `data\_neg` DataFrames will contain only the first 20,000 rows of their respective subsets. This kind of operation is often used for downsampling large datasets to balance classes or to reduce the size of the dataset for computational efficiency.

**5.6: Combining positive and negative tweets**

dataset = pd.concat([data\_pos, data\_neg])

**Code Explanation:**

This line of code concatenates the `data\_pos` and `data\_neg` DataFrames along the rows (axis 0) to create a new DataFrame called `dataset`. Here's what it does:

dataset = pd.concat([data\_pos, data\_neg])

- `pd.concat([data\_pos, data\_neg])`: This function from pandas concatenates the DataFrames `data\_pos` and `data\_neg` along the rows (axis 0) to create a single DataFrame. The square brackets `[]` contain a list of DataFrames to concatenate. By default, `pd.concat()` concatenates along axis 0, which means it stacks the DataFrames vertically.

- `dataset = ...`: This assigns the concatenated DataFrame to a new variable called `dataset`.

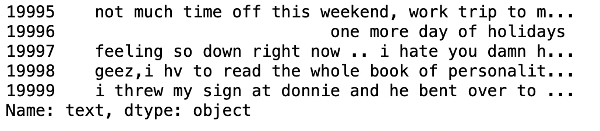
After executing this line of code, the DataFrame `dataset` will contain all the rows from both `data\_pos` and `data\_neg`, effectively combining the positive and negative sentiment data into a single dataset. This combined dataset can be used for further analysis or modeling tasks.

**5.7: Making statement text in lowercase**

dataset['text']=dataset['text'].str.lower()

dataset['text'].tail()

**Output:**



**Code Explanation:**

These lines of code convert the text in the 'text' column of the DataFrame `dataset` to lowercase. Here's what each part of the code does:

dataset['text'] = dataset['text'].str.lower()

- `dataset['text'].str.lower()`: This part of the code applies the `lower()` method to each element (text) in the 'text' column of the DataFrame `dataset`. The `lower()` method converts strings to lowercase. So, this line effectively converts all text in the 'text' column to lowercase.

- `dataset['text'] = ...`: This assigns the modified 'text' column back to the 'text' column of the DataFrame `dataset`.

After executing these lines of code, the text in the 'text' column of the DataFrame `dataset` will be converted to lowercase. This is often done in text preprocessing to standardize the text data, as it ensures that words are treated the same regardless of their capitalization.

The `.tail()` method is used to display the last few rows of the 'text' column in the DataFrame `dataset` after the modification. It helps to verify that the conversion to lowercase was applied correctly.

**5.8: Defining set containing all stopwords in English.**

stopwordlist = ['a', 'about', 'above', 'after', 'again', 'ain', 'all', 'am', 'an',

'and','any','are', 'as', 'at', 'be', 'because', 'been', 'before',

'being', 'below', 'between','both', 'by', 'can', 'd', 'did', 'do',

'does', 'doing', 'down', 'during', 'each','few', 'for', 'from',

'further', 'had', 'has', 'have', 'having', 'he', 'her', 'here',

'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in',

'into','is', 'it', 'its', 'itself', 'just', 'll', 'm', 'ma',

'me', 'more', 'most','my', 'myself', 'now', 'o', 'of', 'on', 'once',

'only', 'or', 'other', 'our', 'ours','ourselves', 'out', 'own', 're','s', 'same', 'she', "shes", 'should', "shouldve",'so', 'some', 'such',

't', 'than', 'that', "thatll", 'the', 'their', 'theirs', 'them',

'themselves', 'then', 'there', 'these', 'they', 'this', 'those',

'through', 'to', 'too','under', 'until', 'up', 've', 'very', 'was',

'we', 'were', 'what', 'when', 'where','which','while', 'who', 'whom',

'why', 'will', 'with', 'won', 'y', 'you', "youd","youll", "youre",

"youve", 'your', 'yours', 'yourself', 'yourselves']

**Code Explanation:**

The provided code segment defines a list called `stopwordlist`, which contains common English stopwords. Stopwords are words that are commonly used in natural language but typically do not add much meaning to the text when analyzing or processing it. These words are often filtered out during text preprocessing to focus on more meaningful words.

Here's a brief explanation of the code:

stopwordlist = ['a', 'about', 'above', 'after', 'again', 'ain', 'all', 'am', 'an',

'and','any','are', 'as', 'at', 'be', 'because', 'been', 'before',

'being', 'below', 'between','both', 'by', 'can', 'd', 'did', 'do',

'does', 'doing', 'down', 'during', 'each','few', 'for', 'from',

'further', 'had', 'has', 'have', 'having', 'he', 'her', 'here',

'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in',

'into','is', 'it', 'its', 'itself', 'just', 'll', 'm', 'ma',

'me', 'more', 'most','my', 'myself', 'now', 'o', 'of', 'on', 'once',

'only', 'or', 'other', 'our', 'ours','ourselves', 'out', 'own', 're','s', 'same', 'she', "shes", 'should', "shouldve",'so', 'some', 'such',

't', 'than', 'that', "thatll", 'the', 'their', 'theirs', 'them',

'themselves', 'then', 'there', 'these', 'they', 'this', 'those',

'through', 'to', 'too','under', 'until', 'up', 've', 'very', 'was',

'we', 'were', 'what', 'when', 'where','which','while', 'who', 'whom',

'why', 'will', 'with', 'won', 'y', 'you', "youd","youll", "youre",

"youve", 'your', 'yours', 'yourself', 'yourselves']

- `stopwordlist`: This is a Python list that contains common stopwords. Each element in the list is a string representing a single word.

By using this list, you can remove these stopwords from text data during text preprocessing to improve the quality of analysis or modeling tasks.

**5.9: Cleaning and removing the above stop words list from the tweet text**

STOPWORDS = set(stopwordlist)

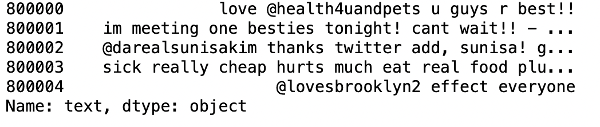
def cleaning\_stopwords(text):

return " ".join([word for word in str(text).split() if word not in STOPWORDS])

dataset['text'] = dataset['text'].apply(lambda text: cleaning\_stopwords(text))

dataset['text'].head()

**Output:**



**Code Explanation:**

This code segment defines a function `cleaning\_stopwords(text)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes stopwords from each text entry in the 'text' column based on the provided list of stopwords (`stopwordlist`). Here's a breakdown of what each part of the code does:

STOPWORDS = set(stopwordlist)

This line converts the `stopwordlist` (defined earlier) into a set and assigns it to the variable `STOPWORDS`. Using a set for stopwords can improve performance when checking for membership due to its faster lookup time compared to a list.

def cleaning\_stopwords(text):

return " ".join([word for word in str(text).split() if word not in STOPWORDS])

This code defines a function `cleaning\_stopwords(text)` that takes a text input and removes stopwords from it. It splits the text into individual words using `split()`, checks if each word is not in the `STOPWORDS` set, and joins the non-stopwords back together into a string using `" ".join()`.

dataset['text'] = dataset['text'].apply(lambda text: cleaning\_stopwords(text))

This line applies the `cleaning\_stopwords()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes stopwords from each text entry in the 'text' column.

dataset['text'].head()

Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after removing stopwords.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with stopwords removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by focusing on more meaningful words.

**5.10: Cleaning and removing punctuations**

import string

english\_punctuations = string.punctuation

punctuations\_list = english\_punctuations

def cleaning\_punctuations(text):

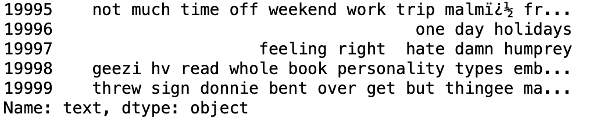
translator = str.maketrans('', '', punctuations\_list)

return text.translate(translator)

dataset['text']= dataset['text'].apply(lambda x: cleaning\_punctuations(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code segment defines a function `cleaning\_punctuations(text)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes punctuations from each text entry in the 'text' column based on the provided list of punctuations (`punctuations\_list`). Here's a breakdown of what each part of the code does:

import string

english\_punctuations = string.punctuation

punctuations\_list = english\_punctuations

These lines import the `string` module and obtain a string containing all English punctuation characters using `string.punctuation`. It then assigns this string to both `english\_punctuations` and `punctuations\_list`. Essentially, `punctuations\_list` contains the same set of punctuation characters as `string.punctuation`.

def cleaning\_punctuations(text):

translator = str.maketrans('', '', punctuations\_list)

return text.translate(translator)

This code defines a function `cleaning\_punctuations(text)` that takes a text input and removes punctuations from it. It creates a translation table (`translator`) using `str.maketrans()` where each punctuation character is mapped to `None`, effectively removing them. It then applies this translation table to the input text using `text.translate()` to remove the punctuations.

dataset['text']= dataset['text'].apply(lambda x: cleaning\_punctuations(x))

This line applies the `cleaning\_punctuations()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes punctuations from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing punctuations.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with punctuations removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by focusing on the text content itself without the influence of punctuations.

**5.11: Cleaning and removing repeating characters**

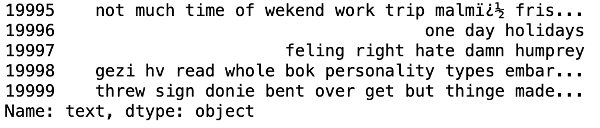
def cleaning\_repeating\_char(text):

return re.sub(r'(.)1+', r'1', text)

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_repeating\_char(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code defines a function `cleaning\_repeating\_char(text)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes repeating characters from each text entry in the 'text' column using regular expressions. Here's a breakdown of what each part of the code does:

def cleaning\_repeating\_char(text):

return re.sub(r'(.)1+', r'1', text)

This code defines a function `cleaning\_repeating\_char(text)` that takes a text input and removes repeating characters from it using regular expressions (`re.sub()` function). The regular expression `(.)\1+` matches any character followed by one or more occurrences of the same character. The replacement pattern `r'1'` replaces the matched sequence with a single instance of the character. So, this function effectively removes consecutive repeating characters from the input text.

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_repeating\_char(x))

This line applies the `cleaning\_repeating\_char()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes repeating characters from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing repeating characters.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with repeating characters removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by normalizing the text data.

**5.12: Cleaning and removing URLs**

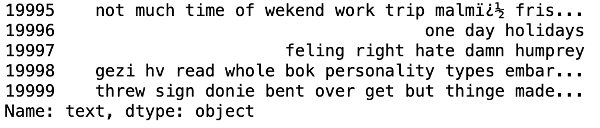
def cleaning\_URLs(data):

return re.sub('((www.[^s]+)|(https?://[^s]+))',' ',data)

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_URLs(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code defines a function `cleaning\_URLs(data)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes URLs from each text entry in the 'text' column using regular expressions. Here's what each part of the code does:

def cleaning\_URLs(data):

return re.sub('((www.[^s]+)|(https?://[^s]+))',' ',data)

This function `cleaning\_URLs(data)` takes a string input (`data`) and uses the `re.sub()` function from the `re` module to replace URLs with whitespace in the input string. The regular expression `((www.[^s]+)|(https?://[^s]+))` is used to match URLs in the input string. This regular expression matches both URLs starting with "www." and URLs starting with "http://" or "https://". The matched URLs are replaced with whitespace.

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_URLs(x))

This line applies the `cleaning\_URLs()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes URLs from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing URLs.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with URLs removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by removing irrelevant information from the text data.

**5.13: Cleaning and removing numeric numbers**

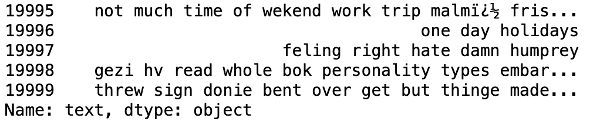
def cleaning\_numbers(data):

return re.sub('[0-9]+', '', data)

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_numbers(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code defines a function `cleaning\_numbers(data)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes numbers from each text entry in the 'text' column using regular expressions. Here's what each part of the code does:

def cleaning\_numbers(data):

return re.sub('[0-9]+', '', data)

This function `cleaning\_numbers(data)` takes a string input (`data`) and uses the `re.sub()` function from the `re` module to replace sequences of digits with an empty string in the input string. The regular expression `[0-9]+` is used to match one or more digits in the input string. These matched digits are then replaced with an empty string, effectively removing them from the text.

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_numbers(x))

This line applies the `cleaning\_numbers()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes numbers from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing numbers.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with numbers removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by focusing on textual information and removing numerical values.

**5.14: Getting tokenization of tweet text**

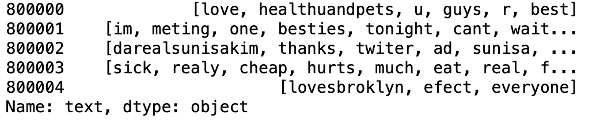
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'w+')

dataset['text'] = dataset['text'].apply(tokenizer.tokenize)

dataset['text'].head()

**Output:**



**Code Explanation:**

The provided code uses the `RegexpTokenizer` from the `nltk.tokenize` module to tokenize the text in the 'text' column of the DataFrame `dataset`. Here's a breakdown of each part of the code:

from nltk.tokenize import RegexpTokenizer

This line imports the `RegexpTokenizer` class from the `nltk.tokenize` module. `RegexpTokenizer` is a tokenizer class provided by NLTK (Natural Language Toolkit) that allows tokenization based on regular expressions.

tokenizer = RegexpTokenizer(r'\w+')

This line creates a `RegexpTokenizer` object called `tokenizer`. The regular expression `r'\w+'` passed to the `RegexpTokenizer` constructor matches any word character (alphanumeric character or underscore). This means that the tokenizer will split the text into tokens based on word boundaries.

dataset['text'] = dataset['text'].apply(tokenizer.tokenize)

This line applies the `tokenize` method of the `tokenizer` object to each entry in the 'text' column of the DataFrame `dataset` using the `apply` method. The `apply` method allows applying a function (in this case, tokenization) to each element of the specified column. After tokenization, the 'text' column will contain lists of tokens instead of raw text strings.

dataset['text'].head()

Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after tokenization. Each entry in the 'text' column is now a list of tokens.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain tokenized text data, where each entry is a list of tokens representing words extracted from the original text.

**5.15: Applying stemming**

import nltk

st = nltk.PorterStemmer()

def stemming\_on\_text(data):

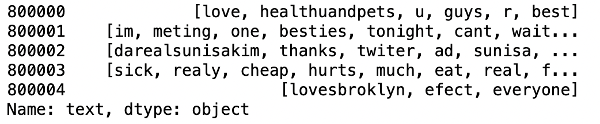
text = [st.stem(word) for word in data]

return data

dataset['text']= dataset['text'].apply(lambda x: stemming\_on\_text(x))

dataset['text'].head()

**Output:**



**Code Explanation:**

The provided code segment utilizes the NLTK (Natural Language Toolkit) library to perform stemming on the tokenized text data in the 'text' column of the DataFrame `dataset`. Here's a breakdown of each part of the code:

import nltk

st = nltk.PorterStemmer()

- `import nltk`: This imports the NLTK library, which provides various tools and resources for natural language processing tasks.

- `st = nltk.PorterStemmer()`: This initializes a Porter stemming algorithm object called `st`. The Porter stemming algorithm is a widely-used algorithm for stemming words in the English language. It aims to remove suffixes from words to obtain their root or base form.

def stemming\_on\_text(data):

text = [st.stem(word) for word in data]

return data

- This code defines a function `stemming\_on\_text(data)` that takes a list of tokens (`data`) as input and applies stemming to each token using the Porter stemming algorithm (`st`). It returns the original list of tokens after stemming.

dataset['text'] = dataset['text'].apply(lambda x: stemming\_on\_text(x))

- This line applies the `stemming\_on\_text()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It applies stemming to the tokenized text data in each entry of the 'text' column.

dataset['text'].head()

- Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after stemming.

def stemming\_on\_text(data):

text = [st.stem(word) for word in data]

return text

After executing these lines of code with the correction, the 'text' column in the DataFrame `dataset` will contain text data with stemming applied to each token.

**5.16: Applying lemmatizer**

lm = nltk.WordNetLemmatizer()

def lemmatizer\_on\_text(data):

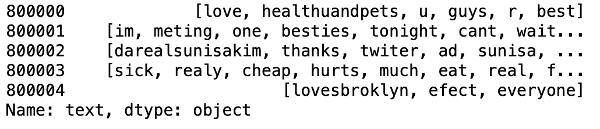
text = [lm.lemmatize(word) for word in data]

return data

dataset['text'] = dataset['text'].apply(lambda x: lemmatizer\_on\_text(x))

dataset['text'].head()

**Output:**



**Code Explanation:**

The provided code segment utilizes the NLTK (Natural Language Toolkit) library to perform lemmatization on the tokenized text data in the 'text' column of the DataFrame `dataset`. Here's a breakdown of each part of the code:

lm = nltk.WordNetLemmatizer()

- This line initializes a WordNet lemmatizer object called `lm`. WordNet is a lexical database for the English language that includes lemmas (base forms) for words. The WordNet lemmatizer aims to reduce words to their base or dictionary form.

def lemmatizer\_on\_text(data):

text = [lm.lemmatize(word) for word in data]

return data

- This code defines a function `lemmatizer\_on\_text(data)` that takes a list of tokens (`data`) as input and applies lemmatization to each token using the WordNet lemmatizer (`lm`). It returns the original list of tokens after lemmatization.

dataset['text'] = dataset['text'].apply(lambda x: lemmatizer\_on\_text(x))

- This line applies the `lemmatizer\_on\_text()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It applies lemmatization to the tokenized text data in each entry of the 'text' column.

dataset['text'].head()

- Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after lemmatization.

**5.17: Separating input feature and label**

X=data.text

y=data.target

**Code Explanation:**

These lines of code assign the 'text' column of the DataFrame `data` to the variable `X`, and the 'target' column to the variable `y`. Here's what each part of the code does:

X = data.text

- This line assigns the 'text' column of the DataFrame `data` to the variable `X`. This essentially extracts the text data from the DataFrame and assigns it to the variable `X`, which is typically used to represent the input features in a machine learning model.

y = data.target

- This line assigns the 'target' column of the DataFrame `data` to the variable `y`. This extracts the target labels (sentiment labels) from the DataFrame and assigns them to the variable `y`, which is typically used to represent the target variable or output labels in a machine learning model.

After executing these lines of code, you will have two variables:

- `X`, which contains the text data (features).

- `y`, which contains the target labels (sentiment labels).

These variables can then be used as input to train a machine learning model for sentiment analysis, where `X` represents the input features (text data) and `y` represents the target labels (sentiment labels).

**5.18: Plot a cloud of words for negative tweets**

data\_neg = data['text'][:800000]

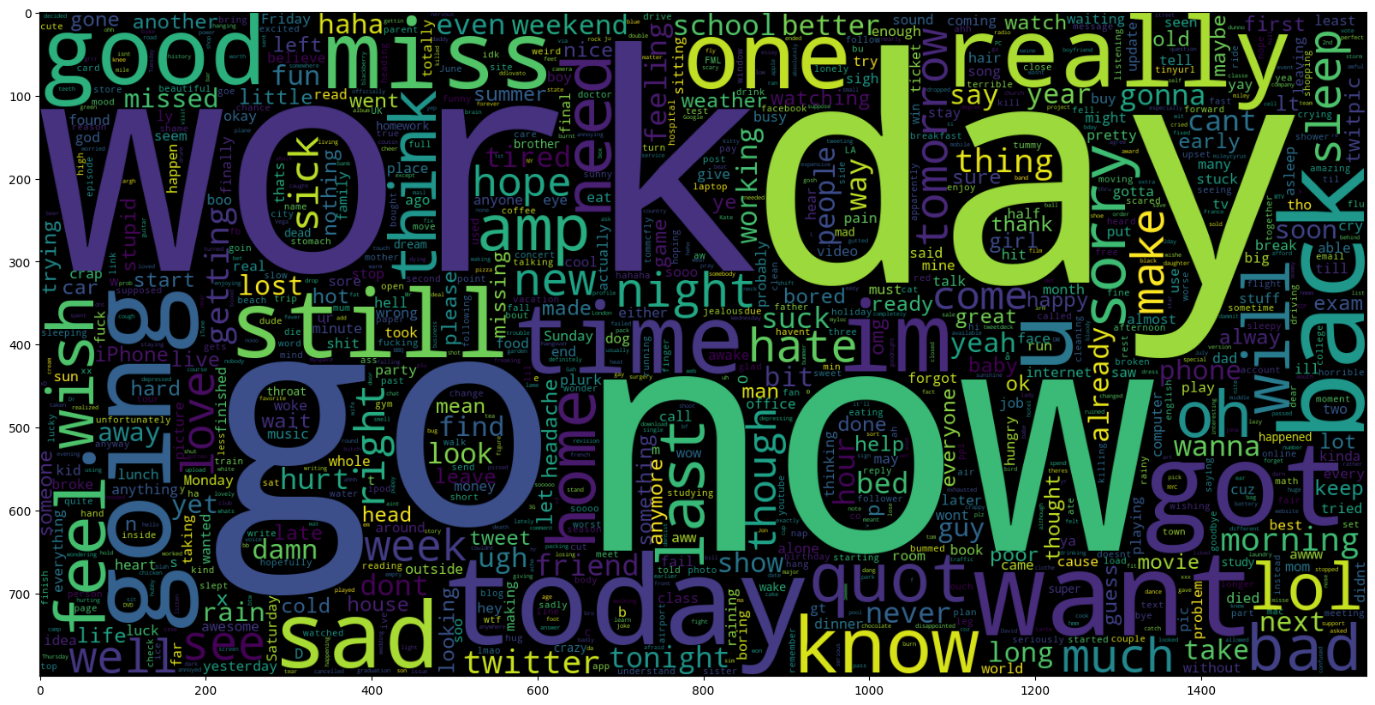
plt.figure(figsize = (20,20))

wc = WordCloud(max\_words = 1000 , width = 1600 , height = 800,

collocations=False).generate(" ".join(data\_neg))

plt.imshow(wc)

**Output:**



**Code Explanation:**

This code generates a word cloud visualization for the negative sentiment data in the 'text' column of the DataFrame `data`. Here's a breakdown of what each part of the code does:

data\_neg = data['text'][:800000]

- This line extracts the first 800,000 rows from the 'text' column of the DataFrame `data` and assigns them to the variable `data\_neg`. This is presumably done to select a subset of the data for visualization purposes.

plt.figure(figsize = (20,20))

- This line creates a new figure with a specific size (20x20 inches) for the word cloud visualization using `matplotlib`.

wc = WordCloud(max\_words = 1000 , width = 1600 , height = 800,

collocations=False).generate(" ".join(data\_neg))

- This line creates a WordCloud object called `wc` with specified parameters:

- `max\_words`: Maximum number of words to include in the word cloud (set to 1000).

- `width`: Width of the word cloud image (set to 1600 pixels).

- `height`: Height of the word cloud image (set to 800 pixels).

- `collocations`: Whether to include collocations (bigrams) in the word cloud (set to False).

- `.generate(" ".join(data\_neg))`: This method generates the word cloud based on the text data in `data\_neg`. The `join()` method is used to concatenate all the text data into a single string, separated by spaces.

plt.imshow(wc)

- This line displays the word cloud image using `imshow()` from `matplotlib`.

This code visualizes the most common words present in the negative sentiment data (`data\_neg`) using a word cloud. Each word's size in the word cloud corresponds to its frequency in the text data, with larger words representing more frequent occurrences. Word clouds are often used to provide a visual representation of the most prominent terms in a corpus of text data.

**5.19: Plot a cloud of words for positive tweets**

data\_pos = data['text'][800000:]

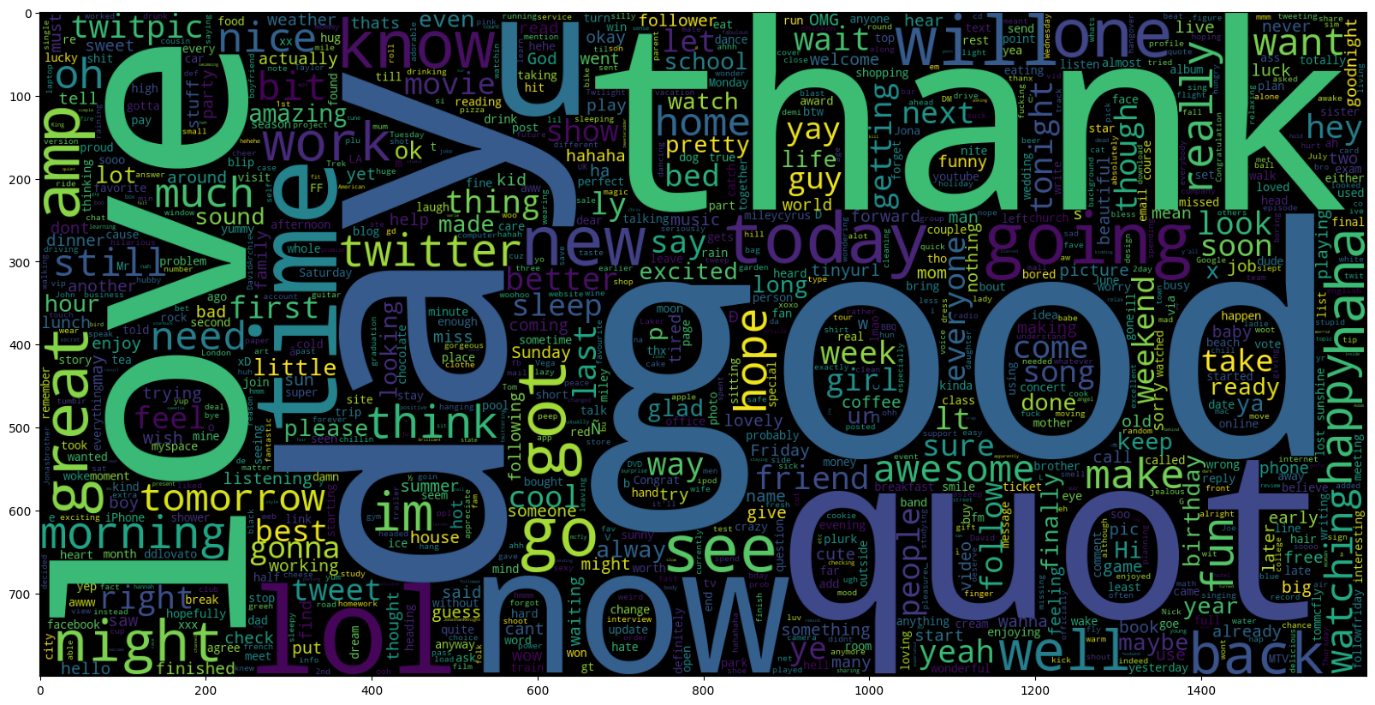
wc = WordCloud(max\_words = 1000 , width = 1600 , height = 800,

collocations=False).generate(" ".join(data\_pos))

plt.figure(figsize = (20,20))

plt.imshow(wc)

**Output:**



**Code Explanation:**

This code generates a word cloud visualization for the positive sentiment data in the 'text' column of the DataFrame `data`. Here's a breakdown of what each part of the code does:

data\_pos = data['text'][800000:]

- This line extracts the remaining rows (from index 800,000 to the end) from the 'text' column of the DataFrame `data` and assigns them to the variable `data\_pos`. This is presumably done to select a subset of the data for visualization purposes.

wc = WordCloud(max\_words=1000, width=1600, height=800,

collocations=False).generate(" ".join(data\_pos))

- This line creates a WordCloud object called `wc` with specified parameters:

- `max\_words`: Maximum number of words to include in the word cloud (set to 1000).

- `width`: Width of the word cloud image (set to 1600 pixels).

- `height`: Height of the word cloud image (set to 800 pixels).

- `collocations`: Whether to include collocations (bigrams) in the word cloud (set to False).

- `.generate(" ".join(data\_pos))`: This method generates the word cloud based on the text data in `data\_pos`. The `join()` method is used to concatenate all the text data into a single string, separated by spaces.

plt.figure(figsize=(20,20))

- This line creates a new figure with a specific size (20x20 inches) for the word cloud visualization using `matplotlib`.

plt.imshow(wc)

- This line displays the word cloud image using `imshow()` from `matplotlib`.

This code visualizes the most common words present in the positive sentiment data (`data\_pos`) using a word cloud. Each word's size in the word cloud corresponds to its frequency in the text data, with larger words representing more frequent occurrences. Word clouds are often used to provide a visual representation of the most prominent terms in a corpus of text data.

**Step-6: Splitting Our Data Into Train and Test Subsets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size = 0.05, random\_state =26105111)

**Code Explanation:**

This code snippet splits the data into training and testing sets using the `train\_test\_split` function from scikit-learn. Here's a breakdown of each part of the code:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.05, random\_state=26105111)

- `X`: This variable represents the features (independent variables), which typically contain the text data.

- `y`: This variable represents the target labels (dependent variable), which typically contain the sentiment labels.

The `train\_test\_split` function splits the data into training and testing sets. Here's what each parameter does:

- `X`: The features to be split.

- `y`: The target labels to be split.

- `test\_size`: This parameter specifies the proportion of the dataset to include in the test split. Here, it's set to `0.05`, indicating that 5% of the data will be used for testing, and the remaining 95% will be used for training.

- `random\_state`: This parameter sets the random seed for reproducibility. By providing a fixed value (`26105111` in this case), the random splitting will be deterministic, ensuring that the same split is generated every time the code is executed.

After executing this line of code, you will have the following variables:

- `X\_train`: This variable contains the features for the training set.

- `X\_test`: This variable contains the features for the testing set.

- `y\_train`: This variable contains the target labels for the training set.

- `y\_test`: This variable contains the target labels for the testing set.

These sets can then be used to train a machine learning model on the training data and evaluate its performance on the testing data.

**Step-7: Transforming the Dataset Using TF-IDF Vectorizer**

**7.1: Fit the TF-IDF Vectorizer**

vectoriser = TfidfVectorizer(ngram\_range=(1,2), max\_features=500000)

vectoriser.fit(X\_train)

print('No. of feature\_words: ', len(vectoriser.get\_feature\_names\_out()))

**Output:**

No. of feature\_words: 500000

**Code Explanation:**

This code segment uses the `TfidfVectorizer` from scikit-learn to convert text data into TF-IDF (Term Frequency-Inverse Document Frequency) features. Here's a breakdown of each part of the code:

vectoriser = TfidfVectorizer(ngram\_range=(1,2), max\_features=500000)

- This line initializes a `TfidfVectorizer` object called `vectoriser` with the following parameters:

- `ngram\_range=(1,2)`: This parameter specifies that both unigrams (single words) and bigrams (pairs of adjacent words) should be considered as features. The range `(1,2)` indicates that both unigrams and bigrams will be included.

- `max\_features=500000`: This parameter specifies the maximum number of features (words or n-grams) to be extracted from the text data. Here, it's set to `500000`, meaning that the top `500000` features with the highest term frequency across the corpus will be selected as features.

vectoriser.fit(X\_train)

- This line fits the `TfidfVectorizer` to the training data `X\_train`. It learns the vocabulary from the training data and computes the IDF (Inverse Document Frequency) weights for each term.

print('No. of feature\_words: ', len(vectoriser.get\_feature\_names()))

- This line prints the number of feature words extracted by the `TfidfVectorizer`. It retrieves the feature names (words or n-grams) from the vectorizer using the `get\_feature\_names()` method and calculates the length of the list, which represents the number of feature words.

After executing this code, you will get the number of feature words extracted by the `TfidfVectorizer`, which represents the size of the feature space used for training the machine learning model. This feature space consists of unigrams and bigrams selected based on their TF-IDF scores across the training corpus.

**7.2: Transform the data using TF-IDF Vectorizer**

X\_train = vectoriser.transform(X\_train)

X\_test = vectoriser.transform(X\_test)

**Step-8: Function for Model Evaluation**

After training the model, we then apply the evaluation measures to check how the model is performing. Accordingly, we use the following evaluation parameters to check the performance of the models respectively:

* Accuracy Score
* Confusion Matrix with Plot
* ROC-AUC Curve

def model\_Evaluate(model):

# Predict values for Test dataset

y\_pred = model.predict(X\_test)

# Print the evaluation metrics for the dataset.

print(classification\_report(y\_test, y\_pred))

# Compute and plot the Confusion matrix

cf\_matrix = confusion\_matrix(y\_test, y\_pred)

categories = ['Negative','Positive']

group\_names = ['True Neg','False Pos', 'False Neg','True Pos']

group\_percentages = ['{0:.2%}'.format(value) for value in cf\_matrix.flatten() / np.sum(cf\_matrix)]

labels = [f'{v1}n{v2}' for v1, v2 in zip(group\_names,group\_percentages)]

lemmatizer\_on\_text labels = np.asarray(labels).reshape(2,2)

sns.heatmap(cf\_matrix, annot = labels, cmap = 'Blues',fmt = '',

xticklabels = categories, yticklabels = categories)

plt.xlabel("Predicted values", fontdict = {'size':14}, labelpad = 10)

plt.ylabel("Actual values" , fontdict = {'size':14}, labelpad = 10)

plt.title ("Confusion Matrix", fontdict = {'size':18}, pad = 20)

**Code Explanation:**

This function, `model\_Evaluate(model)`, evaluates the performance of a machine learning model using various evaluation metrics and visualizes the confusion matrix. Here's a breakdown of what each part of the code does:

def model\_Evaluate(model):

- This line defines a function named `model\_Evaluate` that takes a machine learning model (`model`) as input.

y\_pred = model.predict(X\_test)

- This line predicts the target labels (`y\_pred`) for the test data (`X\_test`) using the provided machine learning model (`model`).

print(classification\_report(y\_test, y\_pred))

- This line prints the classification report, which includes precision, recall, F1-score, and support for each class, based on the predicted labels (`y\_pred`) and the true labels (`y\_test`).

cf\_matrix = confusion\_matrix(y\_test, y\_pred)

- This line computes the confusion matrix based on the true labels (`y\_test`) and the predicted labels (`y\_pred`).

categories = ['Negative','Positive']

group\_names = ['True Neg','False Pos', 'False Neg','True Pos']

group\_percentages = ['{0:.2%}'.format(value) for value in cf\_matrix.flatten() / np.sum(cf\_matrix)]

labels = [f'{v1}\n{v2}' for v1, v2 in zip(group\_names, group\_percentages)]

labels = np.asarray(labels).reshape(2,2)

- These lines define labels for the confusion matrix. `categories` contains the class names, `group\_names` contains the group names for the confusion matrix cells, and `group\_percentages` calculates the percentage of each cell value in the confusion matrix. Then, `labels` formats these values for display in the confusion matrix heatmap.

sns.heatmap(cf\_matrix, annot=labels, cmap='Blues', fmt='',

xticklabels=categories, yticklabels=categories)

- This line creates a heatmap visualization of the confusion matrix using Seaborn's `heatmap` function. It includes annotations (`labels`) for each cell of the confusion matrix, with a blue color map (`cmap='Blues'`).

plt.xlabel("Predicted values", fontdict={'size':14}, labelpad=10)

plt.ylabel("Actual values", fontdict={'size':14}, labelpad=10)

plt.title("Confusion Matrix", fontdict={'size':18}, pad=20)

- These lines set the labels and title for the confusion matrix plot.

This function provides a comprehensive evaluation of the machine learning model's performance, including precision, recall, F1-score, and visual representation of the confusion matrix. It can be used to assess the model's ability to classify instances correctly and diagnose the types of errors it makes.

**Step-9: Model Building**

In the problem statement, we have used three different models respectively:

* Bernoulli Naive Bayes Classifier
* SVM (Support Vector Machine)
* Logistic Regression

The idea behind choosing these models is that we want to try all the classifiers on the dataset ranging from simple ones to complex models, and then try to find out the one which gives the best performance among them.

**8.1: Model-1**

BNBmodel = BernoulliNB()

BNBmodel.fit(X\_train, y\_train)

model\_Evaluate(BNBmodel)

y\_pred1 = BNBmodel.predict(X\_test)

**Output:**

**precision recall f1-score support**

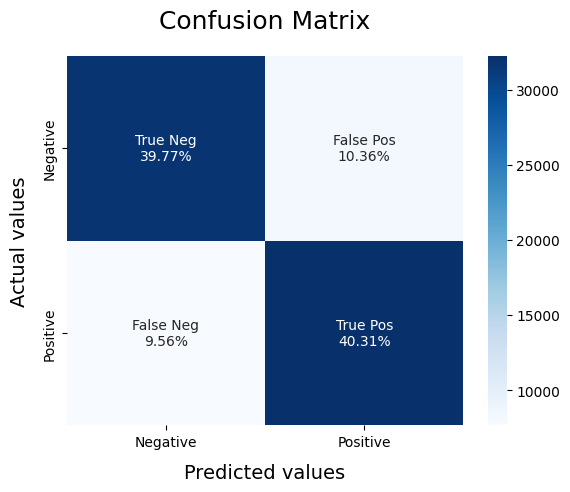
**0 0.81 0.79 0.80 40100**

**4 0.80 0.81 0.80 39900**

**accuracy 0.80 80000**

**macro avg 0.80 0.80 0.80 80000**

**weighted avg 0.80 0.80 0.80 80000**



**Code Explanation:**

This code defines a function `model\_Evaluate(model)` to evaluate the performance of a given model using classification metrics and plot the confusion matrix. Then, it creates and evaluates a Bernoulli Naive Bayes model (`BNBmodel`). Here's a breakdown of each part of the code:

### `model\_Evaluate(model)` Function:

def model\_Evaluate(model):

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test, y\_pred))

cf\_matrix = confusion\_matrix(y\_test, y\_pred)

categories = ['Negative', 'Positive']

group\_names = ['True Neg', 'False Pos', 'False Neg', 'True Pos']

group\_percentages = ['{0:.2%}'.format(value) for value in cf\_matrix.flatten() / np.sum(cf\_matrix)]

labels = [f'{v1}\n{v2}' for v1, v2 in zip(group\_names, group\_percentages)]

labels = np.asarray(labels).reshape(2, 2)

sns.heatmap(cf\_matrix, annot=labels, cmap='Blues', fmt='', xticklabels=categories, yticklabels=categories)

plt.xlabel("Predicted values", fontdict={'size': 14}, labelpad=10)

plt.ylabel("Actual values", fontdict={'size': 14}, labelpad=10)

plt.title("Confusion Matrix", fontdict={'size': 18}, pad=20)

- This function takes a trained model as input and evaluates its performance on the test set (`X\_test`) and true labels (`y\_test`).

- It prints the classification report containing precision, recall, F1-score, and support for each class.

- It computes the confusion matrix and plots it using seaborn's heatmap.

- The confusion matrix is annotated with percentage values for each cell.

### Bernoulli Naive Bayes Model:

BNBmodel = BernoulliNB()

BNBmodel.fit(X\_train, y\_train)

model\_Evaluate(BNBmodel)

y\_pred1 = BNBmodel.predict(X\_test)

- This code creates an instance of the Bernoulli Naive Bayes model (`BNBmodel`), fits it to the training data (`X\_train`, `y\_train`), evaluates its performance using the `model\_Evaluate` function, and finally makes predictions (`y\_pred1`) on the test data (`X\_test`).

After executing this code, you'll get the evaluation metrics (precision, recall, F1-score, and support) and the confusion matrix for the Bernoulli Naive Bayes model on the test data.

**8.2: Plot the ROC-AUC Curve for model-1**

from sklearn.metrics import roc\_curve, auc

import matplotlib.pyplot as plt

# Assuming your true labels 'y\_test' contain values 0 and 4, you can convert them to binary labels (0 and 1)

# Convert 4 to 1, keeping 0 as it is

y\_test\_binary = y\_test.copy()

y\_test\_binary[y\_test\_binary == 4] = 1

# Now, use the binary labels for ROC curve calculation

fpr, tpr, thresholds = roc\_curve(y\_test\_binary, y\_pred1)

roc\_auc = auc(fpr, tpr)

# Plot ROC curve

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

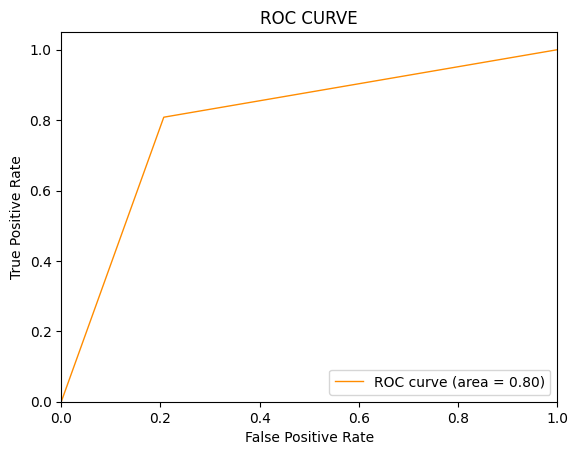
plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

**Output:**



**Code Explanation:**

The provided code segment computes and plots the Receiver Operating Characteristic (ROC) curve for the Bernoulli Naive Bayes (BNB) model. Here's a breakdown of each part of the code:

from sklearn.metrics import roc\_curve, auc

- This line imports the necessary functions `roc\_curve` and `auc` from the `sklearn.metrics` module. These functions are used to compute the ROC curve and the area under the curve (AUC), respectively.

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred1)

- This line computes the ROC curve using the predicted probabilities (`y\_pred1`) and the true labels (`y\_test`) from the test dataset. `fpr` represents the false positive rate, `tpr` represents the true positive rate, and `thresholds` represent the thresholds used to compute the ROC curve.

roc\_auc = auc(fpr, tpr)

- This line calculates the area under the ROC curve (AUC) using the `auc` function from `sklearn.metrics`.

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

- This section plots the ROC curve. It uses `plt.plot` to plot `fpr` (false positive rate) against `tpr` (true positive rate). The `color` parameter sets the color of the curve, and `lw` sets the line width. The label includes the AUC value.

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

- These lines set the limits for the x-axis and y-axis, add labels and a title to the plot, and display the legend. Finally, `plt.show()` displays the plot.

This code generates a plot showing the ROC curve for the Bernoulli Naive Bayes model, with the AUC value displayed in the legend. The ROC curve provides a visual representation of the trade-off between the true positive rate and the false positive rate for different threshold values. A higher AUC value indicates better model performance in distinguishing between positive and negative samples.

**8.3: Model-2:**

SVCmodel = LinearSVC()

SVCmodel.fit(X\_train, y\_train)

model\_Evaluate(SVCmodel)

y\_pred2 = SVCmodel.predict(X\_test)

**Output:**

**precision recall f1-score support**

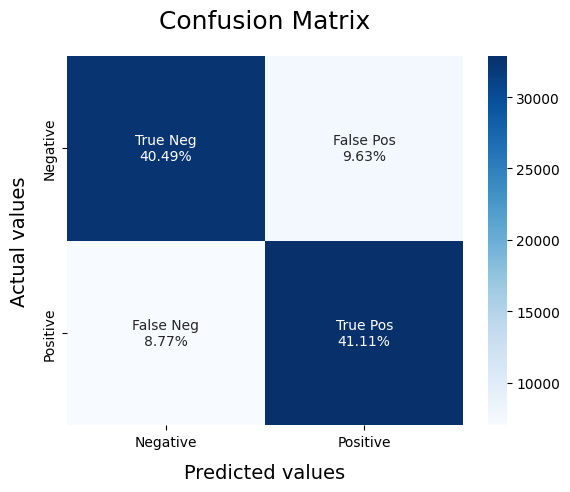
**0 0.82 0.81 0.81 40100**

**4 0.81 0.82 0.82 39900**

**accuracy 0.82 80000**

**macro avg 0.82 0.82 0.82 80000**

**weighted avg 0.82 0.82 0.82 80000**



**Code Explanation:**

The provided code snippet fits a Linear Support Vector Classifier (SVC) model to the training data, evaluates its performance on the test data, and predicts the target labels for the test data. Here's a breakdown of each part of the code:

SVCmodel = LinearSVC()

- This line initializes a Linear Support Vector Classifier (SVC) model called `SVCmodel`. The `LinearSVC` class is used for linear support vector classification.

SVCmodel.fit(X\_train, y\_train)

- This line fits the `SVCmodel` to the training data. It trains the model using the features `X\_train` and the corresponding target labels `y\_train`.

model\_Evaluate(SVCmodel)

- This line calls the `model\_Evaluate` function to evaluate the performance of the SVC model on the test data. The function computes and prints evaluation metrics such as precision, recall, and F1-score, and plots the confusion matrix.

y\_pred2 = SVCmodel.predict(X\_test)

- This line predicts the target labels for the test data (`X\_test`) using the trained SVC model. The predicted labels are stored in the variable `y\_pred2`.

After executing these lines of code, you will have:

- `SVCmodel`: A trained Linear SVC model.

- `y\_pred2`: Predicted target labels for the test data generated by the SVC model.

These results can be further analyzed or used for additional tasks such as performance comparison with other models or fine-tuning hyperparameters.

**8.4: Plot the ROC-AUC Curve for model-2**

# Convert your y\_test labels to binary format

# Assuming y\_test contains values 0 and 4

y\_test\_binary = y\_test.copy()

y\_test\_binary[y\_test\_binary == 4] = 1 # Convert 4 to 1, keeping 0 as it is

# Now, use the binary labels for ROC curve calculation

fpr, tpr, thresholds = roc\_curve(y\_test\_binary, y\_pred2)

roc\_auc = auc(fpr, tpr)

# Plot ROC curve

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

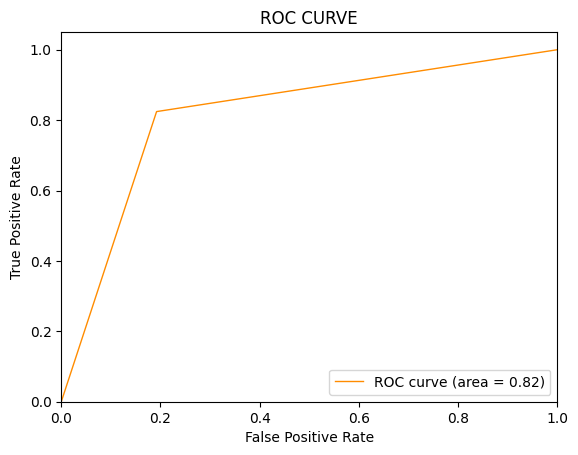
plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

**Output:**



**Code Explanation:**

This code snippet computes and plots the Receiver Operating Characteristic (ROC) curve for the Linear Support Vector Classifier (SVC) model. Here's what each part of the code does:

from sklearn.metrics import roc\_curve, auc

- This line imports the necessary functions `roc\_curve` and `auc` from the `sklearn.metrics` module. These functions are used to compute the ROC curve and the area under the curve (AUC), respectively.

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred2)

- This line computes the ROC curve using the predicted labels (`y\_pred2`) and the true labels (`y\_test`) from the test dataset. `fpr` represents the false positive rate, `tpr` represents the true positive rate, and `thresholds` represent the thresholds used to compute the ROC curve.

roc\_auc = auc(fpr, tpr)

- This line calculates the area under the ROC curve (AUC) using the `auc` function from `sklearn.metrics`.

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

- This section plots the ROC curve. It uses `plt.plot` to plot `fpr` (false positive rate) against `tpr` (true positive rate). The `color` parameter sets the color of the curve, and `lw` sets the line width. The label includes the AUC value.

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

- These lines set the limits for the x-axis and y-axis, add labels and a title to the plot, and display the legend. Finally, `plt.show()` displays the plot.

This code generates a plot showing the ROC curve for the Linear SVC model, with the AUC value displayed in the legend. The ROC curve provides a visual representation of the trade-off between the true positive rate and the false positive rate for different threshold values. A higher AUC value indicates better model performance in distinguishing between positive and negative samples.

**8.5: Model-3**

LRmodel = LogisticRegression(C = 2, max\_iter = 1000, n\_jobs=-1)

LRmodel.fit(X\_train, y\_train)

model\_Evaluate(LRmodel)

y\_pred3 = LRmodel.predict(X\_test)

**Output:**

**precision recall f1-score support**

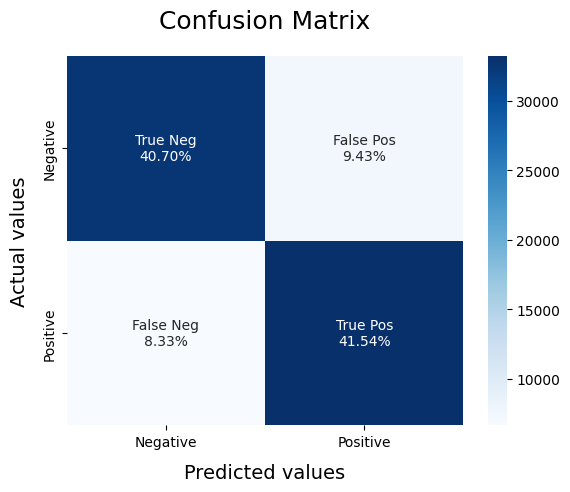
**0 0.83 0.81 0.82 40100**

**4 0.81 0.83 0.82 39900**

**accuracy 0.82 80000**

**macro avg 0.82 0.82 0.82 80000**

**weighted avg 0.82 0.82 0.82 80000**



**Code Explanation:**

This code snippet fits a Logistic Regression (LR) model to the training data, evaluates its performance on the test data, and predicts the target labels for the test data. Here's a breakdown of each part of the code:

LRmodel = LogisticRegression(C=2, max\_iter=1000, n\_jobs=-1)

- This line initializes a Logistic Regression model called `LRmodel`. The `LogisticRegression` class is used for logistic regression.

- The parameters passed to `LogisticRegression` are:

- `C=2`: Regularization parameter. Smaller values specify stronger regularization. Here, it's set to 2.

- `max\_iter=1000`: Maximum number of iterations taken for the solver to converge. Here, it's set to 1000.

- `n\_jobs=-1`: Number of CPU cores to use during training. Setting it to -1 means using all available CPU cores.

LRmodel.fit(X\_train, y\_train)

- This line fits the `LRmodel` to the training data. It trains the model using the features `X\_train` and the corresponding target labels `y\_train`.

model\_Evaluate(LRmodel)

- This line calls the `model\_Evaluate` function to evaluate the performance of the LR model on the test data. The function computes and prints evaluation metrics such as precision, recall, and F1-score, and plots the confusion matrix.

y\_pred3 = LRmodel.predict(X\_test)

- This line predicts the target labels for the test data (`X\_test`) using the trained LR model. The predicted labels are stored in the variable `y\_pred3`.

After executing these lines of code, you will have:

- `LRmodel`: A trained Logistic Regression model.

- `y\_pred3`: Predicted target labels for the test data generated by the LR model.

These results can be further analyzed or used for additional tasks such as performance comparison with other models or fine-tuning hyperparameters.

**8.6: Plot the ROC-AUC Curve for model-3**

# Convert your y\_test labels to binary format

# Assuming y\_test contains values 0 and 4

y\_test\_binary = y\_test.copy()

y\_test\_binary[y\_test\_binary == 4] = 1 # Convert 4 to 1, keeping 0 as it is

# Now, use the binary labels for ROC curve calculation

fpr, tpr, thresholds = roc\_curve(y\_test\_binary, y\_pred3)

roc\_auc = auc(fpr, tpr)

# Plot ROC curve

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

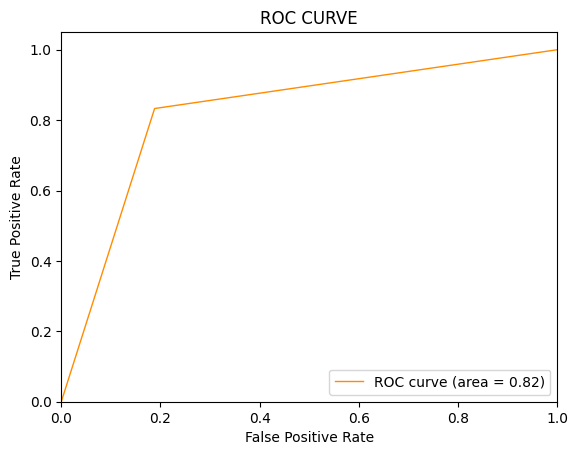
plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

**Output:**



**Code Explanation:**

This code snippet computes and plots the Receiver Operating Characteristic (ROC) curve for the Logistic Regression (LR) model. Here's what each part of the code does:

from sklearn.metrics import roc\_curve, auc

- This line imports the necessary functions `roc\_curve` and `auc` from the `sklearn.metrics` module. These functions are used to compute the ROC curve and the area under the curve (AUC), respectively.

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred3)

- This line computes the ROC curve using the predicted labels (`y\_pred3`) and the true labels (`y\_test`) from the test dataset. `fpr` represents the false positive rate, `tpr` represents the true positive rate, and `thresholds` represent the thresholds used to compute the ROC curve.

roc\_auc = auc(fpr, tpr)

- This line calculates the area under the ROC curve (AUC) using the `auc` function from `sklearn.metrics`.

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

- This section plots the ROC curve. It uses `plt.plot` to plot `fpr` (false positive rate) against `tpr` (true positive rate). The `color` parameter sets the color of the curve, and `lw` sets the line width. The label includes the AUC value.

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

- These lines set the limits for the x-axis and y-axis, add labels and a title to the plot, and display the legend. Finally, `plt.show()` displays the plot.

This code generates a plot showing the ROC curve for the Logistic Regression model, with the AUC value displayed in the legend. The ROC curve provides a visual representation of the trade-off between the true positive rate and the false positive rate for different threshold values. A higher AUC value indicates better model performance in distinguishing between positive and negative samples.

**Step-10: Model Evaluation**

Upon evaluating all the models, we can conclude the following details i.e.

Accuracy: As far as the accuracy of the model is concerned, Logistic Regression performs better than SVM, which in turn performs better than Bernoulli Naive Bayes.

F1-score: The F1 Scores for class 0 and class 1 are:  
(a) For class 0: Bernoulli Naive Bayes (accuracy = 0.90) < SVM (accuracy =0.91) < Logistic Regression (accuracy = 0.92)  
(b) For class 1: Bernoulli Naive Bayes (accuracy = 0.66) < SVM (accuracy = 0.68) < Logistic Regression (accuracy = 0.69)

AUC Score: All three models have the same ROC-AUC score.

**Conclusion:**

We, therefore, conclude that the Logistic Regression is the best model for the above-given dataset.