# Determining the Authenticity of Job Posts on LinkedIn

## Classification Project


**Members:**

Rawan Ibrahim

Eslam Shouman

Mohamed Mahmoud

Sarah Ahmed


**Under the supervision of:**

TA / Omar Sherif

# Table of Contents

**Overview:**

The project focuses on addressing the challenge of distinguishing between real and fake job postings on LinkedIn. By leveraging advanced machine learning techniques and conducting thorough data analysis, we aim to develop a robust classification model that can accurately identify fraudulent job listings, ensuring a trustworthy and reliable experience for job seekers.

Key Features and Accomplishments:

1. **Data Analysis and Visualization:** In the initial phase of the project, we conducted data analysis by applying various statistical techniques and visualizations to gain insights into the characteristics and patterns of both real and fake job postings. Plotting the data allowed us to identify key trends and potential differentiators, providing valuable information for subsequent stages of the project.

2. **Interactive Dashboard:** To facilitate data exploration and decision-making, we developed an interactive dashboard using Power BI. This dashboard provides an intuitive interface for users to explore the dataset, visualize important metrics, and obtain insights from the classification model's predictions.

3. **Data Preprocessing:** We implemented various preprocessing techniques to clean and prepare the dataset for analysis. This included handling missing values, removing duplicate entries, and standardizing text data for effective feature extraction.

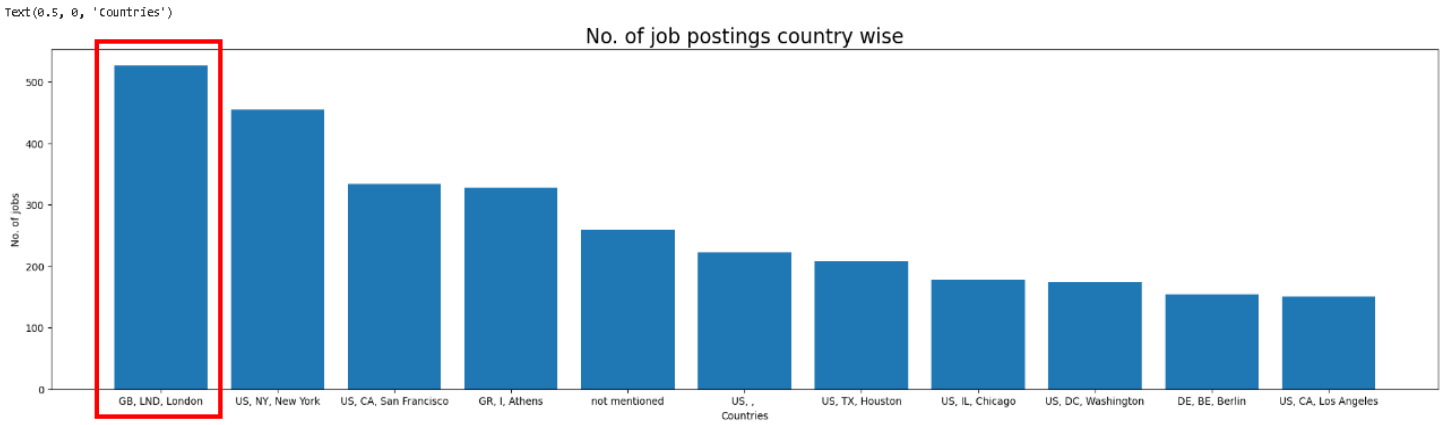
4. **Model Development and Evaluation:** Multiple machine learning models were implemented and trained on the preprocessed dataset. Through rigorous experimentation and validation, we identified the best-performing model with an impressive validation accuracy of 0.95. This model was further evaluated on a test dataset, achieving an accuracy of 0.91, demonstrating its effectiveness in real-world scenarios.

5. **Deployment:** The classification model was deployed, allowing for seamless integration with LinkedIn's job posting system or any other relevant platform. This deployment ensures that the model can be readily used to evaluate the authenticity of job postings in real-time, enhancing the overall user experience.

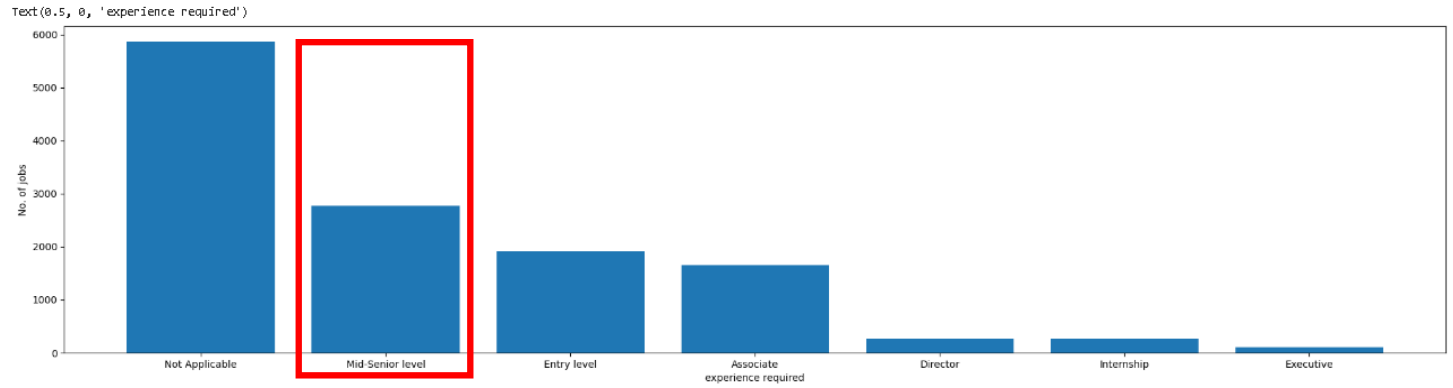
By conducting comprehensive data analysis and visualization, implementing advanced data preprocessing techniques, and developing a high-performing classification model, our project provides an efficient solution to tackle the issue of fake job postings on LinkedIn. With a high accuracy rate, our model helps maintain the integrity and credibility of job listings, creating a trustworthy environment for job seekers.

## Data Analysis and Visualization

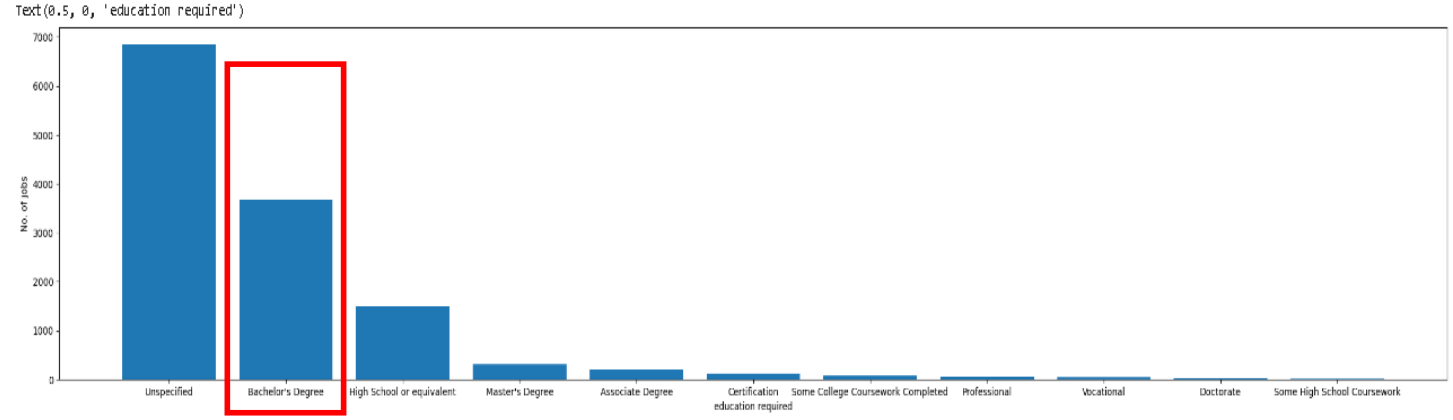
- This visualization provides a clear overview of the number of job postings in each country, highlighting that the highest number of jobs were found in (GB- LND-London). By examining the bar chart, it becomes evident that London exhibits the highest job activity among the countries analyzed.



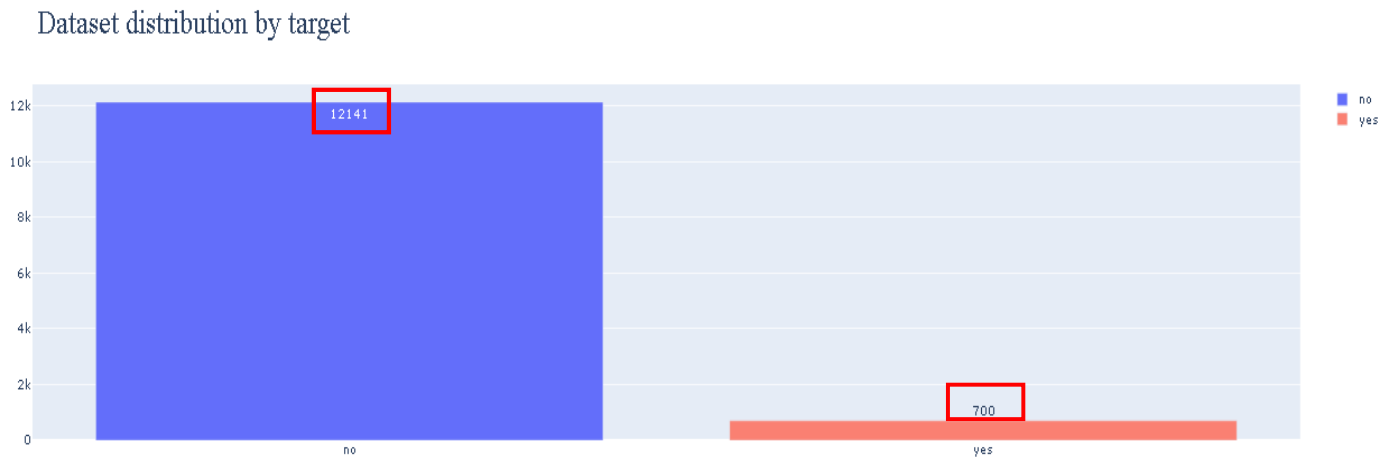

- In addition to analyzing job postings by country, we also explored the distribution of experience requirements for the available positions. The visualization reveals that the "Mid-senior level" experience requirement category had the highest number of job postings.



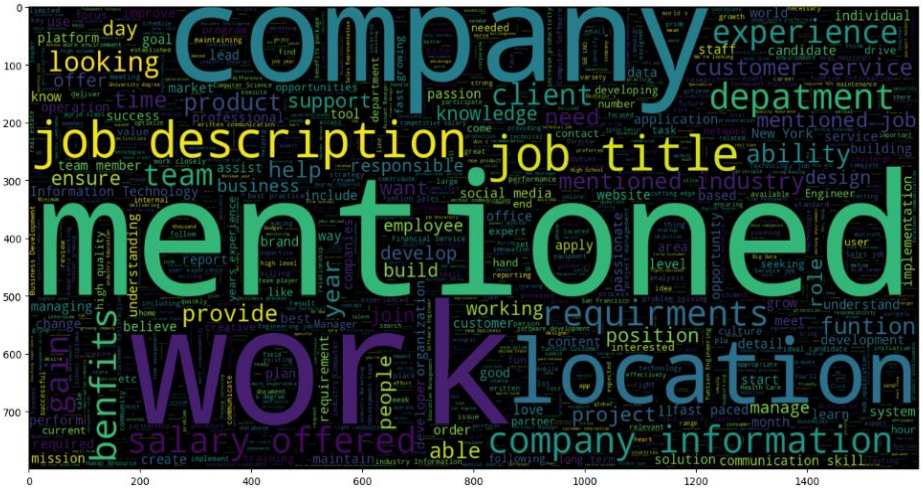

- We further investigated the education requirements for job postings to gain insights into the desired qualifications. The visualization highlights that the highest number of job postings required a "Bachelor's degree" as the preferred educational qualification



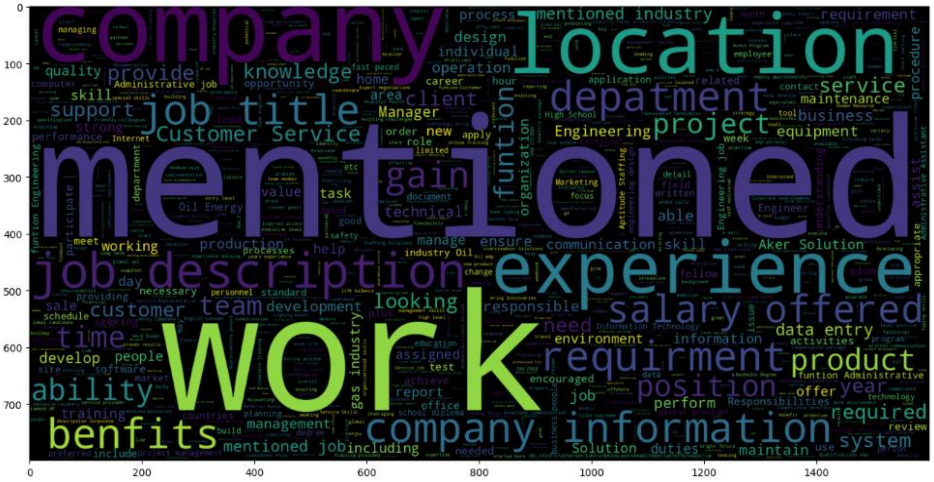
- This visualization provides a clear representation of the distribution of the target variable in the dataset, showcasing the count of 'no' and 'yes' occurrences.




- This word cloud visualization provides a visual representation of the most frequently occurring words in the **real** job postings.

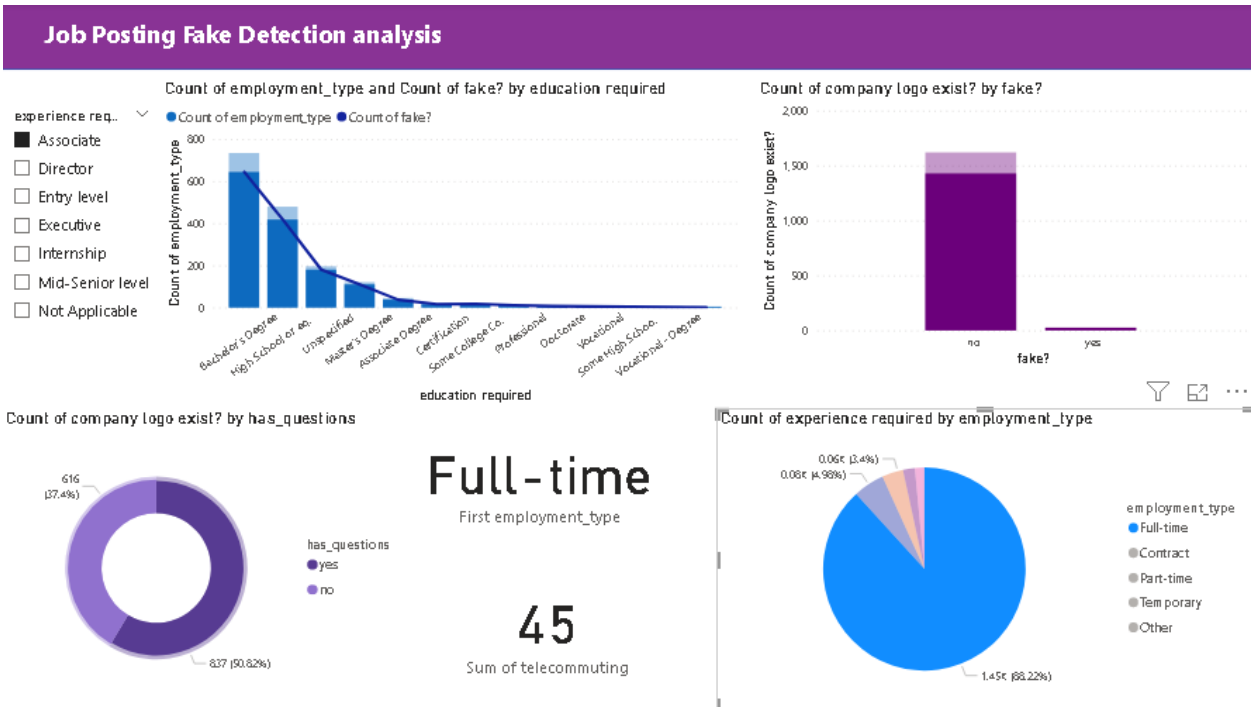



- This word cloud visualization provides a visual representation of the most frequently occurring words in the **fake** job postings.



## Interactive Dashboard

Using Power BI, we developed an interactive dashboard that provided a user-friendly interface for exploring the dataset and visualizing important metrics. Furthermore, Users could obtain predictions from the classification model directly through the dashboard, aiding in real-time decision-making and assessment of job posting authenticity.




## Data Preprocessing

To prepare the dataset for model training, we implemented several preprocessing techniques:

### 1. Dealing with missing values:

During the preprocessing phase, several steps were undertaken to handle missing values in the dataset. Initially, the percentage of missing values in each column was calculated to assess the extent of messiness.

After examining the unique values present in the dataset, it was determined that the majority of missing values could be appropriately filled with the label **"not mentioned."** However, it's important to note that there were three specific columns where a different approach was deemed more suitable.

| | |
|---|---|
| office location | 0.925 |
| department | 29.501 |
| salary offered for the job | 38.401 |
| company information | 8.590 |
| job requirements | 6.932 |
| benefits | 18.733 |
| employment_type | 9.029 |
| experience required | 18.165 |
| education required | 20.904 |
| industry | 12.665 |
| function | 16.640 |

To handle the missing values in these specific columns, the following values were used for filling:

- For the **"employment_type"** column, missing values were filled with the label **"Other"**.
- In the **"experience required"** column, missing values were filled with **"Not Applicable"**.
- Similarly, for the **"education required"** column, missing values were filled with **"Unspecified"**.

```
df["employment_type"].fillna(value="Other", inplace=True)
df["experience required"].fillna(value="Not Applicable", inplace=True)
df["education required"].fillna(value="Unspecified", inplace=True)
```

```
df.fillna(value="not mentioned", inplace=True)
```

```
#CHECKING the precentage of NULL VALUES
nulls=df.isnull().sum()
nulls = round(nulls[nulls!=0] / 27999*100, 3)
nulls
```

```
Series([], dtype: float64)
```

## 2. Combining Columns

After consolidating the text data into a single column and creating a new unified text feature, the original text columns were dropped. This step was taken to eliminate redundancy and reduce dimensionality, focusing solely on the newly created text column. Furthermore, only the columns with a data type of 'int64' were retained, ensuring that the dataset primarily consisted of relevant numeric data.

```
df["Full job"] = "job title: " + df["job title"] + " - the location of the company is " + df["office location"] + " -
```

```
df.drop("job title" ,axis = 1 , inplace = True)
df.drop("office location" ,axis = 1 , inplace = True)
df.drop("department" ,axis = 1 , inplace = True)
df.drop("salary offered for the job" ,axis = 1 , inplace = True)
df.drop("company information" ,axis = 1 , inplace = True)
df.drop("job description" ,axis = 1 , inplace = True)
df.drop("job requirements" ,axis = 1 , inplace = True)
df.drop("benefits" ,axis = 1 , inplace = True)
df.drop("industry" ,axis = 1 , inplace = True)
df.drop("function" ,axis = 1 , inplace = True)
```

## 3. Encoding

We applied priority encoding to three columns: "employment_type," "experience required," and "education required." Priority encoding is a technique used to transform categorical variables into numeric representations based on their relative importance or priority within each category. By applying priority encoding to these specific columns, I aimed to capture and encode the inherent order or hierarchy present in the categorical

values. This encoding approach can be beneficial for certain machine learning algorithms that require numerical inputs.

```
df["employment_type"].replace(to_replace="Other",value=0,inplace=True)
df["employment_type"].replace(to_replace="Temporary",value=1,inplace=True)
df["employment_type"].replace(to_replace="Part-time",value=2,inplace=True)
df["employment_type"].replace(to_replace="Full-time",value=3,inplace=True)
df["employment_type"].replace(to_replace="Contract",value=4,inplace=True)
```

```
df["experience required"].replace(to_replace="Not Applicable",value=0,inplace=True)
df["experience required"].replace(to_replace="Internship",value=1,inplace=True)
df["experience required"].replace(to_replace="Entry level",value=2,inplace=True)
df["experience required"].replace(to_replace="Mid-Senior level",value=3,inplace=True)
df["experience required"].replace(to_replace="Associate",value=4,inplace=True)
df["experience required"].replace(to_replace="Executive",value=5,inplace=True)
df["experience required"].replace(to_replace="Director",value=6,inplace=True)
```

### 4. Handling concatenated words

This "**separate_concatenated_words**" function can be useful for cases where words have been concatenated without proper spacing, allowing for improved readability and subsequent analysis of the text.

```
def separate_concatenated_words(text):
    if isinstance(text, str):
        # Split concatenated words with white space
        words = re.findall(r'\w+', text)
        separated_words = [re.sub('([a-z])([A-Z])', r'\1 \2', word) for word in words]
        return ' '.join(separated_words)
    else:
        return text
```

### 5. Text Preprocessing

The "**Preprocess_text**" function is designed to perform several preprocessing steps on a given text:

```
def preprocess_text(text):
    if isinstance(text, str):
        # Remove any URLs or email addresses
        text = re.sub(r'http\S+|www\S+|https\S+', '', text)
        text = re.sub(r'\S+@\S+', '', text)

        # Remove any sequence of alphanumeric characters starting with 'URL'
        text = re.sub(r'URL\w+', '', text)

        # Remove any non-alphanumeric characters
        text = re.sub(r'[^a-zA-Z0-9\s]', '', text)

        # Split the text into sentences
        sentences = sent_tokenize(text)

        preprocessed_sentences = []
        for sentence in sentences:
            words = nltk.word_tokenize(sentence)

            # Remove any word contains numbers
            words = [word for word in words if not re.match('\w*\d\w*', word)]

            # Remove any word that is a stop word except "not", "Not", "IT", and "it"
            stop_words = set(nltk.corpus.stopwords.words('english'))
            stop_words.discard('not')
            stop_words.discard('Not')
            stop_words.discard('it')
            stop_words.discard('IT')
            words = [word for word in words if word.lower() not in stop_words or word.lower() in {'not', 'it'}]

            preprocessed_sentence = ' '.join(words)
            preprocessed_sentences.append(preprocessed_sentence)

        # Join the preprocessed sentences back together into a single string
        cleaned_text = ' '.join(preprocessed_sentences)

        return cleaned_text
```

- Removal of URLs and email addresses: Any occurrences of URLs or email addresses are removed from the text.
- Removal of 'URL' followed by alphanumeric characters: Any sequences of alphanumeric characters starting with 'URL' are removed.
- Removal of non-alphanumeric characters: All non-alphanumeric characters are removed from the text.
- **Sentence tokenization:** The text is split into individual sentences using the NLTK library's "**sent_tokenize**" function.
- **Word tokenization and filtering:** Each sentence is further tokenized into words using the NLTK library's "**word_tokenize**" function. Words containing numbers are removed, and stop words are filtered out except for specific exceptions.
- **Joining preprocessed sentences:** The preprocessed sentences are rejoined into a single string, separated by white spaces.

The "**Preprocess_text**" function includes an important step to ensure that the word "**not**" is not removed during the stop word removal process. Although "**not**" is typically considered a stop word, it can significantly affect the meaning and sentiment of a text. Therefore, it is preserved in the preprocessed text to maintain the intended message and avoid altering the context of the text. By specifically excluding "**not**" from the list of discarded stop words, the function ensures that this crucial negation term is retained, allowing for accurate analysis and interpretation of the text.

### 6. Whitespace Removal

The "**remove_extra_whitespace**" function proves valuable in text preprocessing tasks where removing redundant whitespace is crucial for data cleanliness and consistency.

```
def remove_extra_whitespace(text):
    if isinstance(text, str):
        # Replace one or more whitespace characters with a single space
        text = re.sub(r'\s+', ' ', text)

        return text.strip()  # Remove leading and trailing whitespace
    else:
        return text
```

### 7. Lowercasing Text Data

The "**lowercase_text**" function is useful for achieving consistency and normalization in text data.

```
def lowercase_text(df):
    for col in df.columns:
        if df[col].dtype == 'object':
            df[col] = df[col].str.lower()
    return df
```

### 8. Removing Duplicate Words (Except 'not')

The "**remove_duplicates**" function ensures that duplicate words are eliminated while preserving the presence of the word **'not'**. This is important as **'not'** carries negation and can significantly impact the meaning and context of the text. By removing duplicate words except for **'not'**, the function helps maintain the integrity and accuracy of the text data for subsequent analysis and processing tasks.

```
def remove_duplicates(text):
    if isinstance(text, list):
        word_list = text
    elif isinstance(text, str):
        word_list = text.split()
    else:
        return text

    unique_words = []
    for i, word in enumerate(word_list):
        if word not in unique_words:
            unique_words.append(word)
        elif i == 0:
            continue
        else:
            unique_words[-1] += ' ' + word
```

### 9. Removing Nonsensical Words

The "**remove_nonsense_words**" function is designed to remove nonsensical words from the given input text. The function utilizes regular expressions to match and remove various patterns of nonsensical words.

```
def remove_nonsense_words(text):
    # Define a list of regular expressions to match non-sensical words
    nonsense_patterns = [
        r'ou{2,}',  # Match "ou" followed by 2 or more "u"s
        r'[bcdfghjklmnpqrstvwxyz]{4,}',  # Match any sequence of 4 or more consonants
        r'[aeiouy]{4,}',  # Match any sequence of 4 or more vowels
        r'[^\x00-\x7F]+'  # Match any non-ASCII characters
        r'[a-z]*(wi){3,}[a-z]*',        # Match any word with "wi" repeated 3 or more times
        r'[a-z]*(x{n}o)+[a-z]*',        # Match any word with "xno" repeated one or more times
        r'[a-z]*(e{2,})+[a-z]*',        # Match any word with "ee" repeated two or more times
        r'n{2,}(e{2,})+[jkn]+',          # Match any word with "n" repeated two or more times followed by "e" repeated two or more times
        r'\b\w*(\w)\1{2,}\w*\b'  # Match any word containing the same character repeated 3 or more times consecutively
    ]

    # Combine the patterns into a single regular expression
    pattern = '|'.join(nonsense_patterns)

    # Use the regular expression to remove the nonsense words
    return re.sub(pattern, '', text)
```

### 10. Enhancing Text Data Quality and Relevance:

Language Detection, Misspelled Word Correction, and Nonsensical/Non-English Word Removal

By applying a series of preprocessing functions, additional checks were performed to ensure the quality and relevance of the text data. The following steps were undertaken:

- **Language Detection:** Initially, a language detection function was implemented to identify if there were any languages other than English present in the text. The function examined the text and returned the indices of the words that belonged to non-English languages.

```
def detect_non_english_words(df, column_name):
    """
    Detect non-English words in a column of a DataFrame.
    """
    non_english_indices = []
    non_english_words = []
    for i, text in enumerate(df[column_name]):
        lang, confidence = langid.classify(text)
        if lang != 'en':
            words = text.split()
            for word in words:
                if langid.classify(word)[0] != 'en' and not is_common_english_word(word):
                    non_english_indices.append(i)
                    non_english_words.append((i, word))
    return non_english_indices, non_english_words
```

- **Misspelled Word Correction:** The focus then shifted to addressing misspelled words within the text. Using appropriate libraries or algorithms, the misspelled words were automatically corrected, improving the accuracy and clarity of the text.

```
def correct_spelling(words):
    """
    Correct spelling of words using autocorrect library.
    """
    sp = autocorrect.Speller(lang='en')
    corrected_words = []
    for word_tuple in words:
        word = word_tuple[1]
        corrected_word = sp(word)
        if corrected_word == word:
            corrected_words.append(word_tuple)
        else:
            corrected_words.append((word_tuple[0], corrected_word))
    return corrected_words
```

- **Removal of Nonsensical and Non-English Words:** Finally, all the remaining words that were either nonsensical or non-English were removed from the text. This step aimed to eliminate any content that could negatively impact the analysis or modeling processes.

By implementing these checks and corrections, the text data was refined and purified, ensuring a more accurate and meaningful analysis of the job postings on LinkedIn.

## 11. Removing Single Letters

- **Remove Single Letters (except ‘e’)**

In this step, single letters in the text are removed to improve text coherence. However, an exception is made for the letter 'e' to ensure it is not mistakenly removed, as it may have a meaningful impact on the next word during subsequent processing.

The function iterates over each word in the text and checks if it consists of a single letter. If a word is identified as a single letter and it is not 'e', it is replaced with an empty string, effectively removing it from the text. This process eliminates single-letter distractions while preserving the significance of the letter 'e' when it is part of a meaningful word.

By removing single letters, except for 'e', the text becomes more coherent and maintains the integrity of subsequent word combinations. This step contributes to the overall effectiveness of the text preprocessing pipeline, preparing the dataset for further analysis and modeling.

```
def remove_single_letters(text):
    # split the text into words
    words = text.split()

    # loop over the words
    for i in range(len(words)):
        # loop over the letters in the word
        j = 0
        while j < len(words[i]):
            if len(words[i]) == 1 and words[i][j] != 'e':
                # if the word has only one letter and it's not 'e', remove it
                words[i] = ''
                break
            else:
                j += 1

    # join the words back into a string
    new_text = ' '.join(words)
```

- **Concatenating "e" with the Next Word**

To enhance the text coherence and maintain the intended meaning, a specific function was developed to concatenate the letter **"e"** with the following word. This step is performed after removing single letters from the text, ensuring that **"e"** remains meaningful in its respective context.

```
# define the function to concatenate "e" with the next word
def concatenate_e(words):
    new_words = []
    for i in range(len(words)):
        if words[i] == "e" and i < len(words) - 1:
            new_words.append(words[i] + words[i+1])
        elif i > 0 and words[i-1] == "e":
            continue
        else:
            new_words.append(words[i])
    new_text = " ".join(new_words)
    return new_text
```

**Text Normalization**

**(Lemmatization)**

In order to further refine the textual data, a **lemmatization** process was applied. The function "**lemmatize_text**"utilizes the WordNet Lemmatizer from the NLTK library to transform words into their base or dictionary form (lemmas).

The function takes a text as input and first splits it into sentences using sentence tokenization. Then, each word in each sentence is lemmatized using the WordNet Lemmatizer. Lemmatization aims to reduce words to their base forms, such as converting "running" to "run" or "better" to "good," thus ensuring consistency and reducing noise in the data.

The lemmatized words are then joined back together into lemmatized sentences, which are finally combined into a single string, maintaining the original sentence structure.

By performing lemmatization, the text data is standardized and optimized for subsequent analysis, allowing for more accurate feature extraction, modeling, and interpretation.

```
def lemmatize_text(text):
    lemmatizer = WordNetLemmatizer()
    if isinstance(text, str):
        # Split the text into sentences
        sentences = sent_tokenize(text)

        # Lemmatize each word in each sentence
        lemmatized_sentences = []
        for sentence in sentences:
            words = nltk.word_tokenize(sentence)
            lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
            lemmatized_sentences.append(' '.join(lemmatized_words))

        # Join the lemmatized sentences back together into a single string
        lemmatized_text = ' '.join(lemmatized_sentences)
        return lemmatized_text
    else:
        return text
```

**Feature Extraction**

**(Vectorization)**

After performing preprocessing steps on the dataset, including text cleaning and transformation, the next step was to apply **vectorization** to convert the text data into numerical representations. The pre.Vectorization function was utilized for this purpose.

The function pre.Vectorization takes the Data Frame as input and specifies the target column name as "**target_name**", which in this case is **'fake?'** This ensures that the target column is properly handled during the vectorization process.

The vectorization process transforms the text data into a format that can be used by machine learning algorithms. It typically involves creating a numerical representation of each text instance, such as using bag-of-words or TF-IDF (Term Frequency-Inverse Document Frequency) techniques.

The resulting "**df_cleaned**" Data Frame contains the vectorized representation of the text column, along with the other columns from the original dataset. This enables the use of the transformed data for further analysis and modeling, such as applying cross-validation techniques.

```
df_cleaned = pre.Vectorization(df , target_name = 'fake?')
df_cleaned
```

After applying vectorization to the text column, the "**df_cleaned**" Data Frame now includes two additional columns: one for positive words and one for negative words. These columns represent the numerical representation of sentiment or polarity associated with the text data. They provide valuable information for sentiment analysis, classification, and regression tasks. The positive words column captures positive sentiment, while the negative words column captures negative sentiment. These columns enhance the Data Frame for further analysis, modeling, and understanding of sentiment in the text data.

| | telecommuting | company_logo | exist? | has_questions | employment_type | experience required | education required | fake? | Job Info._pos | Job Info._neg |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 3 | 0 | 9 | 0 | | 161546283 | 3476394819 |
| 1 | 0 | 1 | 1 | 0 | 0 | 9 | 0 | | 32258285 | 694238594 |
| 2 | 0 | 1 | 1 | 0 | 0 | 9 | 0 | | 239181401 | 5146070593 |
| 3 | 0 | 1 | 1 | 3 | 3 | 1 | 0 | | 266627479 | 5741695576 |
| 4 | 0 | 1 | 0 | 3 | 0 | 9 | 0 | | 284418219 | 6127968530 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 12780 | 0 | 1 | 0 | 0 | 0 | 9 | 0 | | 188289921 | 4054339154 |
| 12781 | 0 | 1 | 1 | 3 | 3 | 9 | 0 | | 441178140 | 9495879441 |
| 12782 | 0 | 1 | 1 | 3 | 0 | 0 | 0 | | 127343432 | 2740588446 |
| 12783 | 0 | 0 | 0 | 3 | 0 | 9 | 0 | | 80359659 | 1727173775 |
| 12784 | 0 | 1 | 0 | 3 | 3 | 1 | 0 | | 244329452 | 5254035729 |

**Model Evaluation and Performance Assessment**

In the process of model development, the dataset was divided into three distinct sets: training, validation, and evaluation. The training and validation sets were created using a 70:30 split, where 70% of the data was allocated for training the models and 30% for validating their performance. This partitioning allowed for the iterative refinement and fine-tuning of the models.

```
x = df_cleaned.drop(['fake?'] , axis = 1)
y = df_cleaned['fake?']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3 , random_state = 42)
```

Several classification models, namely **Logistic Regression**, **K-Nearest Neighbors**, **Decision Tree**, **Extra Tree**, **Random Forest**, **and Extra Trees classifiers**, were trained on the training set. Each model underwent a comprehensive evaluation on the validation set to assess its predictive capabilities and measure performance metrics such as accuracy, **F1 score**, **precision**, and **recall**.

The evaluation process involved making predictions on the validation set using each trained model and comparing the predicted values against the actual labels. By calculating these performance metrics, a comprehensive classification report was generated, summarizing the model performance across different evaluation metrics.

The classification report serves as a valuable tool for model selection and decision-making, providing insights into the strengths and weaknesses of each model. It aids in identifying the most suitable model for the task at hand and assists in further optimization and fine-tuning.

| | Accuracy | F1_score | Precision | Recall |
|---|---|---|---|---|
| LogisticRegression() _ Train Details | 95 | 0 | 0 | 0 |
| LogisticRegression() _ Test Details | 94 | 0 | 0 | 0 |
| KNeighborsClassifier() _ Train Details | 95 | 18 | 71 | 10 |
| KNeighborsClassifier() _ Test Details | 94 | 10 | 35 | 6 |
| DecisionTreeClassifier() _ Train Details | 100 | 100 | 100 | 100 |
| DecisionTreeClassifier() _ Test Details | 93 | 38 | 38 | 38 |
| ExtraTreeClassifier() _ Train Details | 100 | 100 | 100 | 100 |
| ExtraTreeClassifier() _ Test Details | 93 | 38 | 38 | 37 |
| RandomForestClassifier() _ Train Details | 100 | 100 | 100 | 100 |
| RandomForestClassifier() _ Test Details | 94 | 39 | 49 | 33 |
| ExtraTreesClassifier() _ Train Details | 100 | 100 | 100 | 100 |
| ExtraTreesClassifier() _ Test Details | 94 | 39 | 43 | 35 |

**Model Selection and Hyper parameter Tuning**

After evaluating the models on the validation set, the next step involved selecting the best model through hyperparameter tuning using grid search. Grid search allowed for an exhaustive search of various hyperparameter combinations for each model. By leveraging cross-validation, the performance of each configuration was assessed, enabling the identification of the optimal model for each algorithm.

- **Naive Bayes**

```
Best hyperparameters: {'var_smoothing': 1e-09}
Best score: 0.9458039977391322
```

- **Logistic Regression**

```
Best Parameters:  {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
Best Score:  0.7484708225522595
```

- **KNN**

```
Best hyperparameters: {'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}
Best accuracy score: 0.9424517301572927
```

- **Random Forest**

```
Best parameters:  {'max_depth': 15, 'min_samples_split': 10, 'n_estimators': 150}
Best score:  0.9551902220584516
```

- **Decision Tree**

```
Best parameters:  {'max_depth': 10, 'max_features': 'log2', 'min_samples_leaf': 2, 'min_samples_split': 5}
Best score:  0.9500500576146594
```

The best model that emerged from the model selection and hyperparameter tuning phase is the **Random Forest Classifier** (RFC) with the following hyperparameter configuration:

Number of estimators: **150**

Maximum depth: **15**

Minimum samples split: **10**

The RFC model, deemed the best performing model, was saved using the **pickle module**. This allows for the model to be conveniently loaded and applied to the test data

```
import pickle
pickle.dump(rfc, open('RF_model_CCS', 'wb'))
```

**Applying Trained Random Forest Classifier to New Data and Evaluating Performance**

The previously trained Random Forest Classifier (RFC) model, which was saved as **'RF_model_CCS**', is now loaded using the "**pickle.load**" function. This allows us to utilize the trained model to make predictions on new data represented by the features in 'x'.

The loaded model is used to predict the classes for the new data, and the resulting predictions are stored in the '**y_pred**' variable.

```
pickled_model = pickle.load(open('RF_model_CCS', 'rb'))
```

To assess the performance of the RFC model on the new data, we calculate the accuracy by comparing the predicted classes ('y_pred') with the actual classes ('y') using the "**accuracy_score**" function.

```
y_pred = pickled_model.predict(x)

print("accuracy of RF': {}".format(accuracy_score(y_pred,y)))
print(metrics.classification_report(y,y_pred))
```

Additionally, we generate a comprehensive classification report using the "classification report" function. This report provides detailed metrics such as precision, recall, F1 score, and support for each class. It offers valuable insights into how well the model performs on the new data.

```
accuracy of RF': 0.9115071919949969
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      3030
           1       0.26      0.38      0.31       168

    accuracy                           0.91      3198
   macro avg       0.61      0.66      0.63      3198
weighted avg       0.93      0.91      0.92      3198
```

**Final Accuracy on the test data: 91%**