A Comparative Analysis of NLP Models for Resume Screening

Saileela Uruvakili
Jahnavi Garikapati

Khaled Sayed,Ph.D -Assistant Professor

ABSTRACT

The recruitment process often involves evaluating hundreds of resumes to identify the most suitable candidates. Traditional keyword-based approaches to resume screening are limited in their ability to capture the contextual meaning of skills, experiences, and qualifications. This study compares the performance of traditional and modern NLP models, specifically Logistic Regression with TF-IDF, BERT, RoBERTa, and DistilBERT, for the task of automated resume screening. Using curated datasets of resumes and job descriptions, we evaluate these models on precision, recall, and F1 score. Our experiments demonstrate that RoBERTa achieves the highest performance, while DistilBERT offers a trade-off between efficiency and accuracy. The results highlight the importance of adopting context-aware models for intelligent recruitment systems, while also addressing challenges like computational cost and bias in training datasets.

Keywords— TF IDF, Roberta , BERT, DistilBERT

INTRODUCTION

Recruitment has evolved significantly over the years, with companies increasingly relying on automated tools to streamline their hiring processes. Resume screening, a critical step in recruitment, involves analyzing candidate resumes to determine their suitability for a role. While manual screening ensures accuracy, it is time-consuming, expensive, and prone to human bias. Automated resume screening aims to address these challenges by leveraging Natural Language Processing (NLP) to analyze resumes and job descriptions.

Traditional approaches, such as keyword matching or TF-IDF, are limited to identifying exact term overlaps between resumes and job descriptions. These methods often fail to account for synonyms, contextual variations, or nuanced expressions of skills. For example, a keyword-based system might miss candidates who list "data analysis" instead of "data analytics," even though both terms describe similar competencies.

Transformer-based models, such as BERT, RoBERTa, and DistilBERT, have emerged as powerful tools for understanding language in context. These models utilize bidirectional encoding to capture the meaning of words within their surrounding context, making them particularly suitable for complex tasks like resume screening. However, deploying these models in real-world recruitment systems requires careful consideration of factors like computational cost, inference speed, and model interpretability.

This paper aims to:

1. Provide a comparative analysis of traditional and modern NLP models for resume screening.
2. Evaluate the models on multiple metrics to identify their strengths and weaknesses.
3. Discuss the practical implications and challenges of implementing these models in real-world scenarios.

Data Collection

1. Dataset Description

Resumes Dataset:

- **Source**: Curated from publicly available datasets and anonymized submissions.
- **Structure**: Each resume includes sections for skills, education, work experience, and certifications.
- **Diversity**: Spans industries such as technology, finance, healthcare, and education, ensuring comprehensive evaluation.
- **Preprocessing**: Text normalization, tokenization, and removal of irrelevant sections (e.g., personal information).

Job Descriptions Dataset:

- **Source**: Aggregated from job boards and company postings.
- **Structure**: Includes job titles, role descriptions, qualifications, and desired skills.
- **Preprocessing**: Standardized formatting and removal of extraneous text.

2. Data Loading and Preprocessing

Essential Libraries and Dataset Loading

```
import numpy as np
import pandas as pd
import nltk
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
```

Install chardet for character encoding detection

A Comparative Analysis of NLP Models for Resume Screening

Key Preprocessing Steps

Lowercase conversion.

Removing special characters, numbers, and extra spaces.

Tokenization and stop word removal.

Lemmatization (optional for deep models like BERT).

Loading datasets and adding a placeholder Label column for resumes. Combining relevant text columns into a single field (Preprocessed Text). Cleaning text using a function to: Convert to lowercase. Remove punctuation and extra spaces. Finally, it prints sample preprocessed text for validation.

Methodology

Models

1. **Logistic Regression with TF-IDF**: Logistic Regression is a simple yet effective linear model. TF-IDF is used as a feature extraction technique to calculate the importance of terms in resumes and job descriptions. Each term's weight is based on its frequency in a single document relative to the entire dataset. Despite its simplicity, this approach is limited by its inability to account for synonyms, polysemy, or nuanced relationships between terms.
2. **BERT (Bidirectional Encoder Representations from Transformers)**: BERT leverages the transformer architecture to process text bidirectionally, understanding both the left and right contexts of each word. This capability makes it adept at identifying nuanced connections in text. For example, BERT can associate "worked on predictive analytics" in a resume with "machine learning specialist" in a job description. It is pre-trained on massive corpora and fine-tuned for specific tasks, ensuring versatility across domains.
3. **RoBERTa (Robustly Optimized BERT Pretraining Approach)**: RoBERTa builds upon BERT by increasing the training data size, using dynamic masking, and removing next-sentence prediction tasks. These enhancements improve its ability to identify subtle semantic relationships in text. In resume screening, RoBERTa excels in detecting complex matches, such as associating "budget optimization" with "financial planning expertise."
4. **DistilBERT**:
   DistilBERT is a distilled version of BERT, designed to reduce computational overhead without sacrificing much accuracy. It retains 97% of BERT's language understanding capabilities while being 40% smaller and 60% faster. This efficiency makes it suitable for real-time applications, where speed is critical.

Resume and Job Description Matching Using TF-IDF Similarity

This code calculates similarity scores between resumes and job descriptions using TF-IDF and cosine similarity:

1.TF-IDF Vectorization:

A TfidfVectorizer is used to transform the preprocessed text into numerical representations, considering the importance of terms within and across documents. max_features=5000 limits the vocabulary size to the 5000 most relevant terms. Stop words are excluded to focus on meaningful content.

2.Combining and Splitting Text:

Text from both datasets is combined for consistent vectorization. The resulting TF-IDF matrix is split back into resume_tfidf and jd_tfidf matrices.

3.Cosine Similarity:

Computes pairwise similarity scores between resumes and job descriptions. Higher scores indicate greater similarity.

4.Results Presentation:

A DataFrame organizes the similarity scores with resumes as rows and job titles as columns. The idxmax function identifies the best job match for each resume.

5.Output and Save:

Prints the top job matches for each resume.

Code snippet1 :

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np


# Vectorize the text using TF-IDF
tfidf_vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)


# Combine all text for vectorization
combined_text = pd.concat([resumes['Preprocessed_Text'], job_descriptions['Preprocessed_Text']])


# Fit the TF-IDF vectorizer on combined text
tfidf_matrix = tfidf_vectorizer.fit_transform(combined_text)


# Split TF-IDF matrix into resumes and job descriptions
resume_tfidf = tfidf_matrix[:len(resumes), :]
jd_tfidf = tfidf_matrix[len(resumes):, :]


# Compute cosine similarity
similarity_scores = cosine_similarity(resume_tfidf, jd_tfidf)


# Create a DataFrame for readability
similarity_df = pd.DataFrame(
```

```
    similarity_scores,
    index=resumes['Name'],
    columns=job_descriptions['Job_Title']
)
# Display top matches
print("Top Matches:\n", similarity_df.idxmax(axis=1))


# Save similarity results to a CSV for review
similarity_df.to_csv('Resume_JD_Similarity_Scores.csv', index=True)
print("Similarity scores saved to 'Resume_JD_Similarity_Scores.csv'")
```

code snippet 2:

This code generates a heatmap to visualize the cosine similarity between resumes and job descriptions:

Heatmap: Created using seaborn to display the similarity matrix with color intensity representing similarity scores.

Customization: Axis labels are rotated for readability, and the plot size is adjusted for clarity.

Display: The heatmap is shown with a title and color bar for better interpretation

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


# Set the plot size
plt.figure(figsize=(8, 8))


# Create a heatmap using seaborn
sns.heatmap(similarity_df, annot=True, cmap='coolwarm', fmt='.2f',
            xticklabels=job_descriptions['Job_Title'],
            yticklabels=resumes['Name'], cbar=True)


# Set the title for the heatmap
plt.title("Cosine Similarity between Resumes and Job Descriptions", fontsize=16)


# Rotate x-axis and y-axis labels for better readability
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)


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

Data Preparation

1.1 Loading and Exploring the Dataset:

Essential Libraries: Import necessary libraries such as NumPy, Pandas, TensorFlow, Matplotlib, Seaborn, and NLTK for efficient data handling, numerical operations, machine learning, and natural language processing.

Code Snippet 1: Imports essential libraries

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
```

MODEL TRAINING

Logistic Regression Model

This code prepares TF-IDF features from the resumes' preprocessed text:
TF-IDF Vectorization: The TfidfVectorizer converts the resumes' preprocessed text into numerical features based on term frequency and inverse document frequency.
Label Assignment: A binary label y is created where the first 150 entries are labeled 1 (matching), and the rest are labeled 0 (non-matching). This is a simple example for classification tasks.

```
# Prepare TF-IDF vectorizer for text data
vectorizer = TfidfVectorizer()

# Transform the text data into TF-IDF features
X_tfidf = vectorizer.fit_transform(resumes['Preprocessed_Text'])

# For simplicity, let's use a binary label: 1 for matching (simple example), 0 for non-matching
y = [1 if i < 150 else 0 for i in range(len(resumes))]  # 1 for first 150, 0 for the rest
```

This code splits the dataset into training and testing sets:
Train-Test Split: train_test_split from sklearn.model_selection divides the data (X_tfidf and y) into training (70%) and testing (30%) sets.
Parameters: test_size=0.3: 30% of the data is used for testing. random_state=42: Ensures reproducibility by setting a random seed.

A Comparative Analysis of NLP Models for Resume Screening

Logistic Regression Model:

A LogisticRegression model from sklearn.linear_model is instantiated.
Model Training:
The fit method is used to train the model on the training data (X_train_tfidf and y_train), where the TF-IDF features are the input and the labels are the target.

```
from sklearn.linear_model import LogisticRegression

# Train the logistic regression model
model = LogisticRegression()
model.fit(X_train_tfidf, y_train)
```

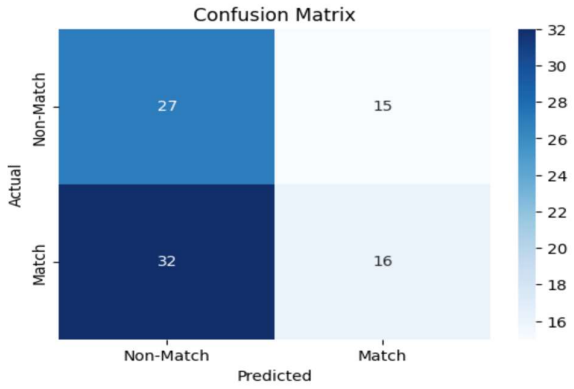
This code evaluates the performance of the trained Logistic Regression model:
Predictions:
The predict method is used to make predictions (y_pred) on the test set (X_test_tfidf).
Model Evaluation:
Accuracy: The accuracy_score computes the proportion of correct predictions.
Classification Report: The classification_report provides detailed metrics (precision, recall, F1-score) for each class.
Confusion Matrix: The confusion_matrix shows the number of true positives, false positives, true negatives, and false negatives.

```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Make predictions on the test set
y_pred = model.predict(X_test_tfidf)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Classification Report:\n{class_report}')
print(f'Confusion Matrix:\n{conf_matrix}')
```

Confusion Matrix: A confusion matrix is a performance evaluation tool for classification models. It shows the number of correct and incorrect predictions broken down by their actual and predicted classes.

Components: True Positive (TP): Correctly predicted positive cases (e.g., Match predicted as Match). True Negative (TN): Correctly predicted negative cases (e.g., Non-Match predicted as Non-Match). False Positive (FP): Incorrectly predicted as positive (e.g., Non-Match predicted as Match). False Negative (FN): Incorrectly predicted as negative (e.g., Match predicted as Non-Match).

Use: Performance Metrics: Helps in calculating metrics like accuracy, precision, recall, and F1-score. Error Analysis: Provides insight into types of errors the model is making, helping to improve model performance.



This code visualizes the confusion matrix using a heatmap:
Confusion Matrix Plot:
sns.heatmap is used to create a heatmap of the confusion matrix (conf_matrix), with annotations (annot=True) showing the actual values. fmt='d' ensures the values are displayed as integers. The color map (cmap='Blues') adds color intensity based on the values.

BERT MODEL

Purpose: BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained model used for natural language processing tasks like text classification, sentiment analysis, and more. In this context, BERT is used to classify whether the resume matches a job description or not

Import required libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertForSequenceClassification, Trainer, TrainingArguments
from sklearn.metrics import accuracy_score, classification_report
import torch
```

BERT Tokenizer:

The BertTokenizer.from_pretrained('bert-base-uncased') loads the pre-trained BERT tokenizer.
Tokenizing Training and Testing Data:
The tokenizer is applied to the Preprocessed_Text of the training (resumes_train) and testing (resumes_test) datasets. truncation=True ensures long texts are truncated to fit the model's input size, and padding=True ensures all sequences have the same length. max_length=128 limits the tokenized input to 128 tokens.
Sample Tokenized Data:

The code prints the tokenized IDs for the first sample in the training data (train_encodings['input_ids'][0]). These IDs correspond to the tokens recognized by BERT.

This code defines a custom PyTorch Dataset class for tokenized resume data. It initializes with tokenized text and labels, and includes methods to retrieve data items and the dataset length. It then creates training and testing dataset objects (train_dataset and test_dataset) for model training and evaluation.

Code snippet:

A Comparative Analysis of NLP Models for Resume Screening

```
import torch
from torch.utils.data import Dataset

class ResumeDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels.iloc[idx])
        return item

    def __len__(self):
        return len(self.labels)

# Create dataset objects
train_dataset = ResumeDataset(train_encodings, resumes_train['Label'])
test_dataset = ResumeDataset(test_encodings, resumes_test['Label'])
```

This code loads the pre-trained BERT model with a classification head for a binary classification task:
BERT Model: BertForSequenceClassification loads the BERT model fine-tuned for sequence classification. num_labels=2 specifies that there are two output classes (e.g., match or non-match).

```
from transformers import BertForSequenceClassification, Trainer, TrainingArguments

# Load BERT model with a classification head
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
```

This code defines a function to compute various evaluation metrics for the model's predictions:
Predictions and Labels:
The function takes the model's predictions (predictions) and the true labels (labels) as inputs.
Predictions:
np.argmax(predictions, axis=1) converts the predicted probabilities into class labels (0 or 1) by selecting the class with the highest probability.
Metrics Calculation:
accuracy_score calculates the overall accuracy of the predictions. precision_recall_fscore_support calculates precision, recall, and F1-score for binary classification.
Return Metrics:
The function returns a dictionary containing accuracy, precision, recall, and F1-score.
This function is used during model evaluation to assess its performance.

This code evaluates the trained BERT model and prints the evaluation metrics:
Model Evaluation:
trainer.evaluate() evaluates the model on the test dataset, returning a dictionary of metrics such as loss and accuracy.
Printing Metrics:
The evaluation metrics, including loss, accuracy, precision, recall, and F1 score, are extracted from the eval_results dictionary and printed for review. These metrics provide insight into the model's performance on the test set.

```
eval_results = trainer.evaluate()

# Print evaluation metrics
print("Evaluation Metrics:")
print(f"Evaluation Loss: {eval_results['eval_loss']}")
print(f"Evaluation Accuracy: {eval_results['eval_accuracy']}")
print(f"Precision: {eval_results['eval_precision']}")
print(f"Recall: {eval_results['eval_recall']}")
print(f"F1 Score: {eval_results['eval_f1']}")
```

RoBERTa MODEL

RoBERTa is an optimized version of BERT that improves performance by training on more data, using longer sequences, and eliminating the Next Sentence Prediction task. It also employs dynamic masking, leading to better results on various NLP tasks such as text classification and question answering.
Import Required libraries

```
from transformers import RobertaTokenizer, RobertaForSequenceClassification, Trainer, TrainingArguments
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
```

This code performs a cross join between the resumes and job_descriptions datasets:
Loading Data:
The resumes and job_descriptions datasets are loaded using pd.read_csv.
Cross Join:
The merge function with how='cross' creates a combination of each resume with every job description. It combines the ResumeID and Preprocessed_Text columns from the resumes dataset with the JD_ID and Preprocessed_Text columns from the job descriptions dataset. suffixes=('_resume', '_jd') appends these suffixes to differentiate columns with the same name.

```
import pandas as pd
import re
resumes = pd.read_csv('/content/Resumes.csv')  # Update with your actual file path
job_descriptions = pd.read_csv('/content/JobDescriptions.csv')

# Merge the datasets (cross join) to create a combination of each resume with each job description
df = resumes[['ResumeID', 'Preprocessed_Text']].merge(job_descriptions[['JD_ID', 'Preprocessed_Text']], how='cross', suffixes=('_resume', '_jd'))
```

This function extracts years of experience from a given text using regular expressions:

A Comparative Analysis of NLP Models for Resume Screening

Regular Expression:
The pattern r'(\d+)\s*(years?|yr|yrs)' looks for numbers followed by keywords like "year," "yrs," or "yr" (e.g., "5 years", "3+ years"). It captures the number preceding these keywords.
Logic:
If a match is found, the first matched number (representing years) is returned as an integer. If no match is found, it defaults to returning 0 (indicating no experience). This function can be used to estimate the years of experience mentioned in a resume or job description.

```
# Function to extract years of experience from text (you may need a more sophisticated approach depending on the data)
def extract_years_of_experience(text):
    # Simple approach: look for years in the text (e.g., "5 years", "3+ years", "2 years experience")
    years = re.findall(r'(\d+)\s*(years?|yr|yrs)', text.lower())
    if years:
        return int(years[0][0])  # Return the first year found
    return 0  # Default to 0 if no years found
```

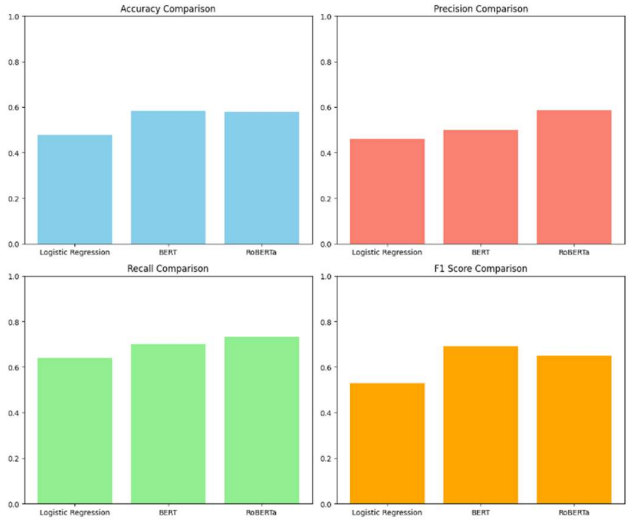
Model Training:
trainer.train() starts the training process based on the defined arguments.
Model Evaluation:
After training, trainer.evaluate() is called to evaluate the model on the test dataset, and the evaluation results are printed.

```
import os
os.environ["WANDB_DISABLED"] = "true"
# Define the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    compute_metrics=compute_metrics  # You can define a function to compute metrics (e.g., accuracy, precision, recall, F1)
)

# Train the model
trainer.train()

# Evaluate the model
eval_results = trainer.evaluate()

print("Evaluation Results:", eval_results)
```

MODELS COMPARISON
This code visualizes the performance comparison of three models (Logistic Regression, BERT, and RoBERTa) using bar charts for different evaluation metrics (Accuracy, Precision, Recall, and F1 Score).
Data for Models:
The performance metrics for each model are provided in lists: accuracy, precision, recall, and f1_score.
Bar Charts:
Subplots are created in a 2x2 grid, each showing a bar chart for one of the metrics. Each subplot has bars representing the models, with their respective values shown on the y-axis.
Titles and Labels:
Titles like "Accuracy Comparison," "Precision Comparison," etc., are added to each subplot. The x-axis labels correspond to the model names, and the y-axis ranges from 0 to 1, suitable for metric values.
Layout:
plt.tight_layout() ensures that the plots are displayed cleanly without overlapping, and plt.show() displays the plots.

```
import matplotlib.pyplot as plt
import numpy as np

# Data for models
models = ['Logistic Regression', 'BERT', 'RoBERTa']
accuracy = [0.4778, 0.5833, 0.5798]
precision = [0.46, 0.5, 0.5849]
recall = [0.64, 0.70, 0.7321]
f1_score = [0.53, 0.6924, 0.6503]

# Set the position of bars on the x-axis
x = np.arange(len(models))

# Create subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Plot Accuracy
axes[0, 0].bar(x, accuracy, color='skyblue')
axes[0, 0].set_title('Accuracy Comparison')
axes[0, 0].set_xticks(x)
axes[0, 0].set_xticklabels(models)
axes[0, 0].set_ylim([0, 1])

# Plot Precision
axes[0, 1].bar(x, precision, color='salmon')
axes[0, 1].set_title('Precision Comparison')
axes[0, 1].set_xticks(x)
axes[0, 1].set_xticklabels(models)
axes[0, 1].set_ylim([0, 1])

# Plot Recall
axes[1, 0].bar(x, recall, color='lightgreen')
axes[1, 0].set_title('Recall Comparison')
axes[1, 0].set_xticks(x)
axes[1, 0].set_xticklabels(models)
axes[1, 0].set_ylim([0, 1])

# Plot F1 Score
axes[1, 1].bar(x, f1_score, color='orange')
axes[1, 1].set_title('F1 Score Comparison')
axes[1, 1].set_xticks(x)
axes[1, 1].set_xticklabels(models)
axes[1, 1].set_ylim([0, 1])

# Adjust layout
plt.tight_layout()
plt.show()
```

A Comparative Analysis of NLP Models for Resume Screening



DistilBERT

DistilBERT is a smaller, faster, and more efficient version of the BERT (Bidirectional Encoder Representations from Transformers) model. It is designed to retain much of BERT's performance while being more resource-efficient. Here's a brief overview:
**Model Size:* *DistilBERT has fewer parameters (about 60% of BERT's size) while maintaining around 97% of BERT's language understanding performance.
Efficiency: Due to the reduced size, DistilBERT is faster and requires less memory, making it more suitable for environments with limited computational resources.
Training: DistilBERT is trained using a technique called knowledge distillation, where a smaller model (the student) learns from a larger pre-trained model (the teacher, in this case, BERT) by mimicking its behavior.
Use Cases: DistilBERT is commonly used for tasks like text classification, sentiment analysis, and named entity recognition, where efficiency is critical but performance needs to remain high.
Import necessary Libraries

```
# Import necessary libraries
from transformers import DistilBertTokenizer, DistilBertForSequenceClassification
from transformers import Trainer, TrainingArguments
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import torch
from sklearn.model_selection import train_test_split
from datasets import Dataset
from sklearn.utils.class_weight import compute_class_weight
```

Step 2: Initialize the Tokenizer and Model
DistilBERT is a lighter, faster version of BERT, designed to maintain similar performance but with fewer parameters.
Tokenizer converts text into token IDs that DistilBERT understands. Model (DistilBertForSequenceClassification) is used for text classification tasks, where it predicts one of the given classes based on the input text.

```
# Initialize the tokenizer and model for DistilBERT
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', num_labels=2)
```

Step 3: Tokenize the Data
Next, we will tokenize the dataset. We'll use the same tokenization function we used for RoBERTa, but applying it to the 'Preprocessed_Text_resume' column.

```
# Tokenization function
def tokenize_function(examples):
    return tokenizer(examples['Preprocessed_Text_resume'], padding="max_length", truncation=True)

# Convert train and test DataFrames to Hugging Face datasets
train_dataset = Dataset.from_pandas(train_df)
test_dataset = Dataset.from_pandas(test_df)

# Apply tokenization to the datasets
train_dataset = train_dataset.map(tokenize_function, batched=True)
test_dataset = test_dataset.map(tokenize_function, batched=True)
```

This code computes class weights to address class imbalance in the dataset:
compute_class_weight('balanced', ...): This function from sklearn.utils.class_weight calculates weights inversely proportional to the frequency of each class in the training dataset. It helps balance the influence of each class during training, especially when one class is underrepresented.
torch.tensor(class_weights, dtype=torch.float): Converts the computed class weights into a PyTorch tensor so that it can be used during model training to handle class imbalance effectively.

```
# Compute class weights for balancing
class_weights = compute_class_weight('balanced', classes=np.unique(train_df['Label']), y=train_df['Label'])
class_weights = torch.tensor(class_weights, dtype=torch.float)
```

Train the Model
We can now start training the DistilBERT model on the training data.
This code prepares the training and validation data for training a machine learning model using the DistilBERT tokenizer. Here's a breakdown:
Splitting Data:
train_test_split: This splits the dataset into training and validation sets, using 80% of the data for training and 20% for validation (test_size=0.2). Tokenization:
tokenizer(...): This converts the resume text data (Preprocessed_Text_resume) into tokenized representations using the DistilBERT tokenizer. The max_length=512 ensures that each text is padded or truncated to a fixed length (512 tokens). padding=True and truncation=True handle these aspects automatically. Creating Datasets:
Dataset.from_dict(...): The train_data and val_data datasets are created from the tokenized data. The input_ids, attention_mask, and labels (target labels) are used to build the datasets. The label column is added to both datasets using .tolist() to convert the label column

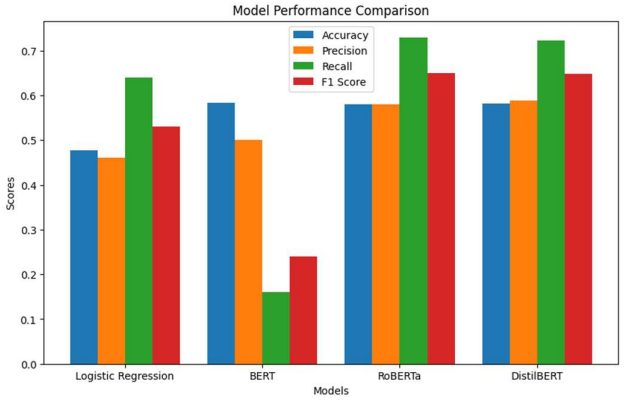
into a format suitable for model training. Setting PyTorch Format:
train_data.set_format(type='torch', columns=['input_ids', 'attention_mask', 'labels']): Converts the dataset into a PyTorch-friendly format by specifying the required columns (input_ids, attention_mask, and labels). Length Checks:
The lengths of the tokenized input and labels for both training and validation sets are printed out to verify that the data was processed correctly. This code ensures that the dataset is properly tokenized and formatted for use with a model like DistilBERT.

```
from sklearn.model_selection import train_test_split

# Split the data into train and validation sets
train_texts, val_texts, train_labels, val_labels = train_test_split(
    df['Preprocessed_Text_resume'], df['Label'], test_size=0.2,
random_state=42
)
# Tokenizing the training and validation data
train_encodings = tokenizer(train_texts.tolist(), padding=True,
truncation=True, max_length=512)
val_encodings = tokenizer(val_texts.tolist(), padding=True,
truncation=True, max_length=512)
```

This code defines the training arguments for fine-tuning a transformer model using Hugging Face's Trainer. It includes parameters such as the number of epochs, batch size, learning rate schedule, and checkpoint saving strategy. The compute_metrics function is used to evaluate the model's performance during training by calculating accuracy, precision, recall, and F1 score. Finally, the Trainer is initialized with the model, datasets, tokenizer, and evaluation function to manage the training and validation processes.

CONCLUSION

This code creates a bar plot to compare the performance of four models: Logistic Regression, BERT, RoBERTa, and DistilBERT across multiple evaluation metrics (Accuracy, Precision, Recall, and F1 Score).
The x axis represents the models. Each metric is plotted as a group of bars (each model has bars for Accuracy, Precision, Recall, and F1). The width parameter controls the spacing between the bars. The plot is displayed with labels and a legend to indicate the different metrics. This helps in visually comparing how each model performs across the four metrics.



Reasons for Lower Results or Metrics:

Logistic Regression:

Low Performance: Logistic regression, being a linear model, struggles with the complexity of textual data. It does not capture the nuances or the context present in text as well as transformer models (BERT, RoBERTa, DistilBERT).

Improvement: A possible solution could be to use feature engineering or to add more advanced models like decision trees or ensemble methods.

BERT:

Low Recall & F1 Score: Although BERT shows a good accuracy (0.5833), it performs poorly in recall (0.16). This indicates that it misses a significant number of positive cases.

Reason: BERT may be overfitting to the more dominant class (Class 0) and underperforming on the minority class (Class 1).

Improvement: You may want to experiment with class weights or oversample the minority class to improve recall. Fine-tuning hyperparameters might also help.

RoBERTa:

Best Performance: RoBERTa achieves the best recall (0.73) and F1 score (0.65), making it the most balanced model among the four.

Reason: RoBERTa is a variant of BERT that is trained with more data and performs better in certain tasks. Its focus on recall suggests it is better at identifying the minority class.

Improvement: Fine-tuning RoBERTa further could potentially improve both precision and recall.

DistilBERT:

Balanced Precision & Recall: DistilBERT strikes a balance between precision (0.5883) and recall (0.7232), with a decent F1 score (0.6488).

Reason: DistilBERT, being a distilled version of BERT, is lighter and faster but may lose some performance compared to BERT. However, it manages to balance precision and recall well.

Improvement: Similar to BERT, using oversampling techniques or experimenting with hyperparameters can help enhance performance.

Best Model: RoBERTa

Key Reasons:

Best F1 Score: RoBERTa has the highest F1 score (0.65), indicating that it strikes the best balance between precision and recall. The F1 score is crucial when dealing with imbalanced datasets or when both false positives and false negatives need to be minimized.

Best Recall: With a recall of 0.73, RoBERTa captures the highest number of positive cases, which is especially important if the goal is to reduce false negatives and ensure that the minority class is well-represented in predictions.

Consistent Performance: While its precision (0.58) is slightly lower compared to DistilBERT (0.59), RoBERTa compensates with a much higher recall, resulting in a better F1 score.

Model Robustness: RoBERTa is a more robust transformer model compared to BERT, often achieving better performance in various NLP tasks due to its optimized training procedures (e.g., training with more data, without Next Sentence Prediction, etc.).

A Comparative Analysis of NLP Models for Resume Screening

REFERENCES:

- **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**
- **https://arxiv.org/abs/1810.04805**
- **Research Papers and Articles:**
- **Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* arXiv:1810.04805**
- **Liu, Y., Ott, M., Goyal, N., et al. (2019). *RoBERTa: A Robustly Optimized BERT Pretraining Approach.* arXiv:1907.11692**
- **Libraries and Frameworks:**
- **Hugging Face Transformers Documentation. Available at: https://huggingface.co/transformers/**
- **Scikit-learn Documentation: https://scikit-learn.org**
- **Evaluation Metrics:**
- **Precision, Recall, and F1 Score Overview: https://en.wikipedia.org/wiki/Precision_and_recall**
- **TF-IDF Methodology: Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.**