Final Project Section5-Team 4

June 17, 2025

[1]:
```
# ensure we're using pytorch
import os
os.environ["TRANSFORMERS_NO_TF"] = "1"
```

This line sets the TRANSFORMERS_NO_TF environment variable to "1", ensuring that Hugging Face Transformers avoids using TensorFlow and instead defaults to PyTorch. This is especially important in environments where both are installed but we want to exclusively use PyTorch.

The environment is now configured to use only PyTorch with the Transformers library.

[37]:
```
# import libraries
import pandas as pd
import re
import torch
import numpy as np
from tqdm import tqdm
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import precision_recall_curve, average_precision_score
from scipy.sparse import hstack
from transformers import AutoTokenizer, AutoModel
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
from wordcloud import WordCloud
from torch.utils.data import Dataset, DataLoader
from transformers import DistilBertForSequenceClassification
from torch.optim import AdamW
from transformers import Trainer, TrainingArguments
from datasets import load_metric
from torch.nn import CrossEntropyLoss
from torch import tensor
```

1 Problem Statement and Approach

In light of many GenAI technologies, there are an increasing number of fake job postings getting posted to our job board website which is causing job seekers loose trust in our website and job postings. Identifying fake job postings will be critical for building user trust as well as customer trust.

Job postings are text heavy and Natural Language Processing (NLP) provides an effective method for assessing job posting content. We'll aim to utilize DistilBERT, a transformer-based model to classify job postings. By fine tuning the DistilBERT model on labeled job posting data, our team will aim to build a classification model to deploy to our job search website to detect fake job postings.

2 Data Understanding (EDA)

With the fake job postings dataset, we will proceed with exploratory data analysis. We'll start by reviewing some initial records to get an idea of how each record is stored and check the columns available in the dataset.

[3]:
```
# Load dataset
df = pd.read_csv('fake_job_postings.csv')
```

[4]:
```
df.head()
```

[4]:
```
   job_id                                      title            location  \
0       1                          Marketing Intern    US, NY, New York
1       2  Customer Service - Cloud Video Production      NZ, , Auckland
2       3    Commissioning Machinery Assistant (CMA)       US, IA, Wever
3       4          Account Executive - Washington DC  US, DC, Washington
4       5                         Bill Review Manager  US, FL, Fort Worth

  department salary_range                                    company_profile  \
0  Marketing          NaN  We're Food52, and we've created a groundbreaki…
1    Success          NaN  90 Seconds, the worlds Cloud Video Production …
2        NaN          NaN  Valor Services provides Workforce Solutions th…
3      Sales          NaN  Our passion for improving quality of life thro…
4        NaN          NaN  SpotSource Solutions LLC is a Global Human Cap…

                                         description  \
0  Food52, a fast-growing, James Beard Award-winn…
1  Organised - Focused - Vibrant - Awesome!Do you…
2  Our client, located in Houston, is actively se…
3  THE COMPANY: ESRI - Environmental Systems Rese…
4  JOB TITLE: Itemization Review ManagerLOCATION:…

                                       requirements  \
0  Experience with content management systems a m…
1  What we expect from you:Your key responsibilit…
```

```
2  Implement pre-commissioning and commissioning …
3  EDUCATION: Bachelor’s or Master’s in GIS, busi…
4  QUALIFICATIONS:RN license in the State of Texa…

                                          benefits  telecommuting  \
0                                              NaN              0
1  What you will get from usThrough being part of…              0
2                                              NaN              0
3  Our culture is anything but corporate-we have …              0
4                           Full Benefits Offered               0

   has_company_logo  has_questions employment_type required_experience  \
0                 1              0           Other          Internship
1                 1              0       Full-time      Not Applicable
2                 1              0             NaN                NaN
3                 1              0       Full-time    Mid-Senior level
4                 1              1       Full-time    Mid-Senior level

  required_education                  industry             function  \
0                NaN                       NaN            Marketing
1                NaN  Marketing and Advertising     Customer Service
2                NaN                       NaN                NaN
3  Bachelor's Degree          Computer Software                Sales
4  Bachelor's Degree     Hospital & Health Care  Health Care Provider

   fraudulent
0           0
1           0
2           0
3           0
4           0
```

[5]:
```
df.columns.tolist()
```

[5]:
```
['job_id',
 'title',
 'location',
 'department',
 'salary_range',
 'company_profile',
 'description',
 'requirements',
 'benefits',
 'telecommuting',
 'has_company_logo',
 'has_questions',
 'employment_type',
```

```
 'required_experience',
 'required_education',
 'industry',
 'function',
 'fraudulent']
```

After analyzing the columns within the dataset, we'll proceed with check for missing values so we can properly handle those missing values while training our model.

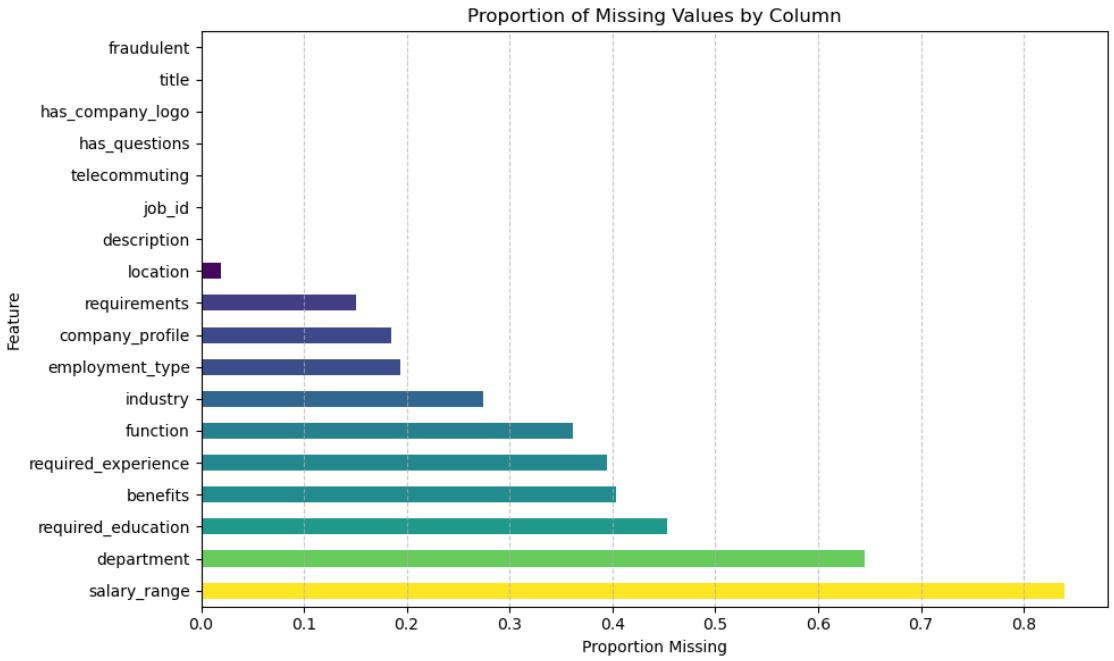
[6]:
```
# check for missing data
missing_data = df.isnull().mean().sort_values(ascending=False)

missing_data
```

[6]:
```
salary_range           0.839597
department             0.645805
required_education     0.453300
benefits               0.403356
required_experience    0.394295
function               0.361018
industry               0.274217
employment_type        0.194128
company_profile        0.185011
requirements           0.150783
location               0.019351
description            0.000056
job_id                 0.000000
telecommuting          0.000000
has_questions          0.000000
has_company_logo       0.000000
title                  0.000000
fraudulent             0.000000
dtype: float64
```

The output highlights columns with significant missing values. Fields with too much missing data might need to be dropped or carefully imputed, depending on their relevance. This informs our data cleaning strategy.

[7]:
```
# adding colors to bars for ease of visualization
colors = plt.cm.viridis(missing_data.values / max(missing_data.values))

# plot missing values
plt.figure(figsize=(10, 6))
missing_data.plot(kind='barh', color=colors)
plt.title('Proportion of Missing Values by Column')
plt.xlabel('Proportion Missing')
plt.ylabel('Feature')
plt.grid(axis='x', linestyle='--', alpha=0.7)
```

```
plt.tight_layout()
plt.show()
```

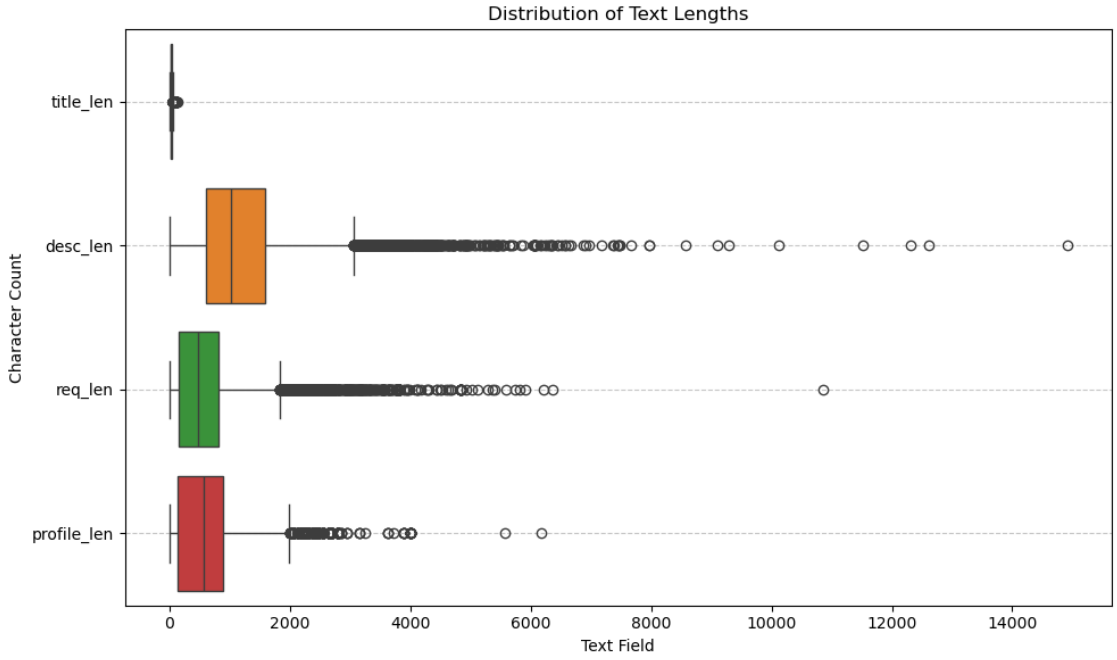


When visualizing the missing values plot, it's much easier to identify which datapoints are missing. We can deduce whether or not we are missing essential components of a job posting and use that to judge whether it's a good idea to drop missing values or apply some imputation as the missing values could prove important for detecting fake jobs. Reviewing the plot, we can see that we're not missing any values from the target class, title, job id, and description which are all important details of a job posting. From the columns with missing data, it is suprising to see missing values for requirements and company profile which we would assume are important elements of a job posting. This could be useful for identifying fake jobs postings.

Now that we know there are some job postings with important information missing, we can check for text length within those fields we believe to be important for real job postings. We decided to inspect job title, job description, job requirements, and company profile to see if there are any strange outliers or suprising insights from what we might expect. We would expect that job description would be the longest text out of the four fields as a quality job posting would ensure to thorougly explain the job requirements.

[8]:
```
# check text length (these can be used as engineered features)
df['title_len'] = df['title'].apply(lambda x: len(str(x)))
df['desc_len'] = df['description'].apply(lambda x: len(str(x)))
df['req_len'] = df['requirements'].apply(lambda x: len(str(x)))
df['profile_len'] = df['company_profile'].apply(lambda x: len(str(x)))

# plot text length
```

```
plt.figure(figsize=(10, 6))
sns.boxplot(data=df[['title_len', 'desc_len', 'req_len', 'profile_len']],␣
 ↪orient='h')
plt.title('Distribution of Text Lengths')
plt.xlabel('Text Field')
plt.ylabel('Character Count')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

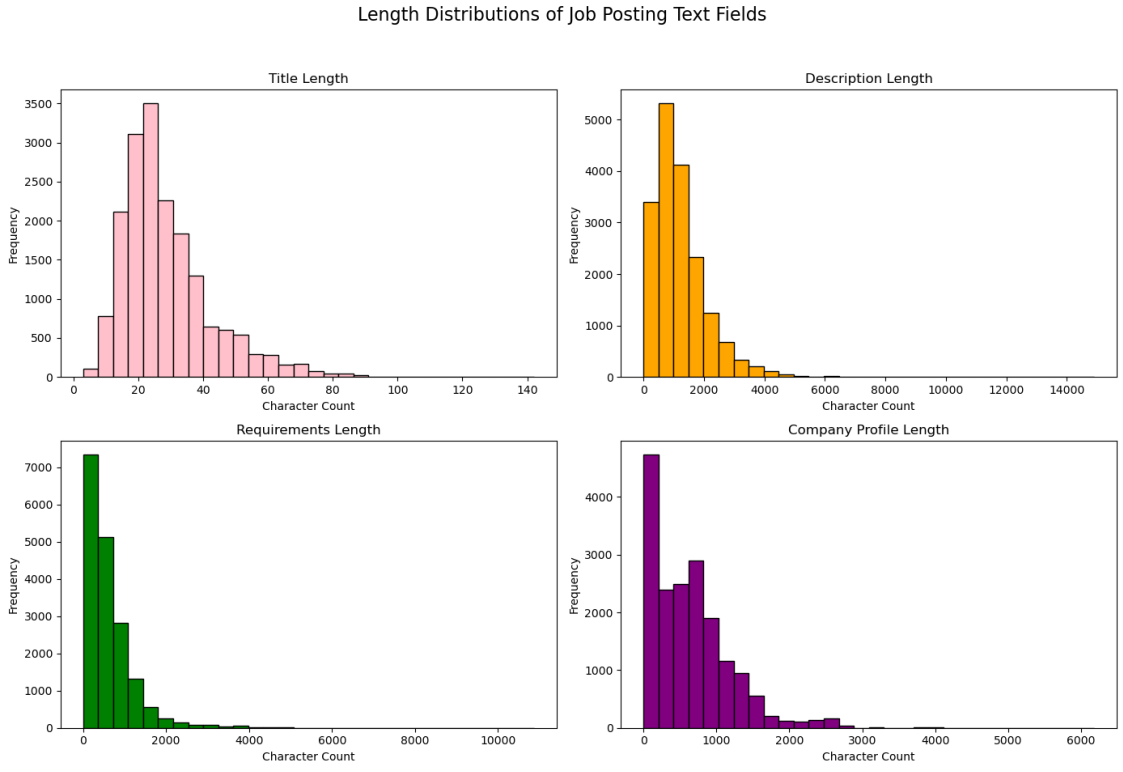


The boxplot provides a clear picture of outliers and the Interquartile Range (IQR) give good guidence on the spread of text length for the important fields. As expected, the job description tends to be the longest text in the job posting with some clear outliers being almost seven times longer than the median job description length. Job titles are relatively short as we would expect with several outliers which may be worth investigating. There are some outliers requirements and profile lengths as well.

It would also be good to get a visualization of the distribution of the text lengths as well.

[9]:
```
# compute field length accounting for missing values
df['title_length'] = df['title'].fillna('').apply(len)
df['description_length'] = df['description'].fillna('').apply(len)
df['requirements_length'] = df['requirements'].fillna('').apply(len)
df['company_profile_length'] = df['company_profile'].fillna('').apply(len)

# plot field length distributions
```

```
fig, axs = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle('Length Distributions of Job Posting Text Fields', fontsize=16)

# Plot histograms
axs[0, 0].hist(df['title_length'], bins=30, edgecolor='black', color='pink')
axs[0, 0].set_title('Title Length')

axs[0, 1].hist(df['description_length'], bins=30, edgecolor='black',␣
 ↪color='orange')
axs[0, 1].set_title('Description Length')

axs[1, 0].hist(df['requirements_length'], bins=30, edgecolor='black',␣
 ↪color='green')
axs[1, 0].set_title('Requirements Length')

axs[1, 1].hist(df['company_profile_length'], bins=30, edgecolor='black',␣
 ↪color='purple')
axs[1, 1].set_title('Company Profile Length')

for ax in axs.flat:
    ax.set_xlabel('Character Count')
    ax.set_ylabel('Frequency')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



This code identifies outliers in text length features using the 95th percentile as a threshold and creates box plots for visualization. The 95th percentile method flags the top 5% of values as potential outliers, which are then stored in binary indicator columns for future use.

The box plots reveal distinct patterns across the four text length features, with title length showing a tight distribution around 30 characters and minimal outliers indicating consistent formatting practices, while description length exhibits the widest variation with a median near 1,000 characters and numerous extreme outliers exceeding 10,000 characters. Requirements length demonstrates moderate spread with a median around 600 characters and some outliers beyond 6,000 characters, and company profile length displays high variability similar to descriptions with a median around 700 characters and outliers extending past 4,000 characters. These results indicate that job descriptions and company profiles have the most variable content length while titles remain consistently concise, with outliers potentially representing either comprehensive job postings or data quality issues that should be addressed in preprocessing.

The text length distribution does provide a clearer picture of these job fields and shows that job descriptions are generally longer than job requirements and company profiles. Nothing suprising with majority of job ittles lower than 60 characters with some outlier job titles being longer than 80 characters. Those may be questionable jobs. The large number of job descriptions and job requirements with character lengths close to zero is alarming which indicates there are many job postings without these fields. We could consider makingn these fields required in order for the job to be posted on our website in order to ensure more quality job posts are being posted for job seekers.

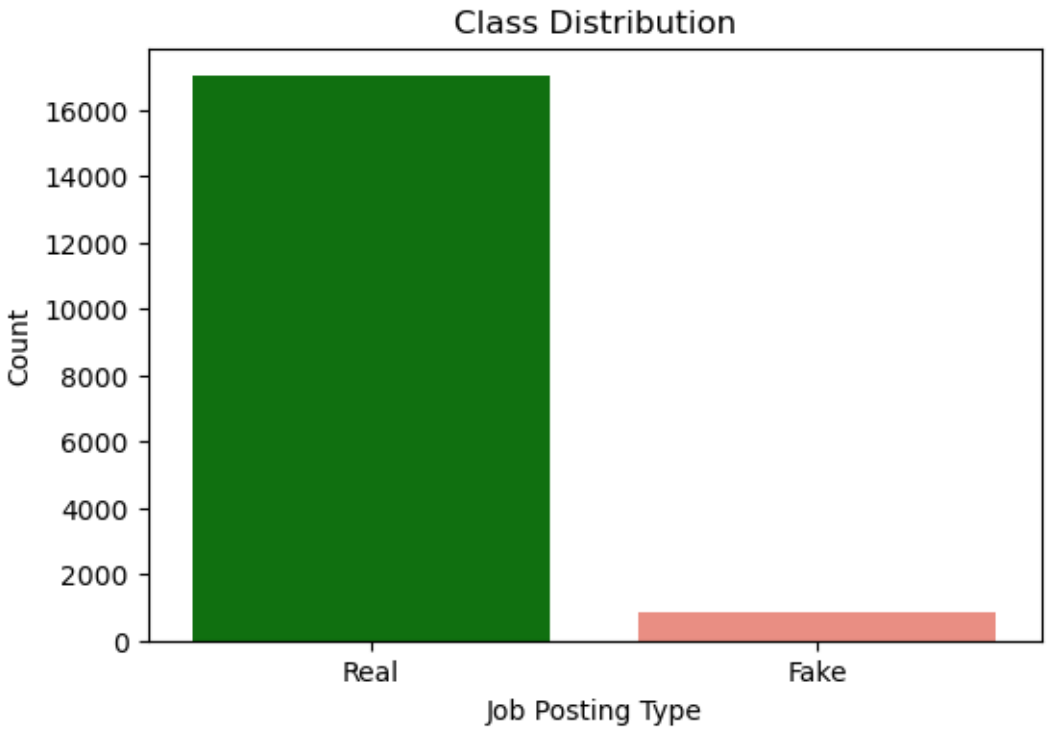
With a good idea of all the columns in the dataset, we'll proceed to check the target class for any imbalances. This will provide a clear picture of how the target class needs to be handled via the model training to ensure imbalances are accounted for.

[10]:
```
# check class balance for fake vs real
class_counts = df['fraudulent'].value_counts()

print(class_counts)
```

```
fraudulent
0    17014
1      866
Name: count, dtype: int64
```

[11]:
```
# set class field to string
df['fraudulent'] = df['fraudulent'].astype(str)

# plot class balance
plt.figure(figsize=(6, 4))
sns.countplot(data=df, x='fraudulent', hue='fraudulent', palette={'0': 'green',␣
 ↪'1': 'salmon'}, legend=False)
plt.title('Class Distribution')
plt.xticks([0, 1], ['Real', 'Fake'])
```

```
plt.xlabel('Job Posting Type')
plt.ylabel('Count')
plt.show()
```



With the number of real job postings of 17014 and the number of fake job postings of 866, this equates to roughly 4.8% of all job postings being fake in our dataset. There is clear class imbalance which is a good thing because we would expect a very small percentage of job postings are fake. This means we will need to account for class imbalance while training our model to ensure there is not an overfitting issue to the majority class. Overfitting can cause accuracy to be extremely inflated and inaccurate and we will need to pay attention to other performance metrics such as precision, recall, and F1-scores.

3 Data Preparation & Feature Engineering

From our exploratory data analysis, we discovered there were missing data for many of the fields in a job posting. For modeling purposes, we will fill in missing values with "Unknown" as missing values from job descriptions or job titles could be a signal of fake jobs.

[12]:
```
# missing values
missing_summary = df.isnull().sum().sort_values(ascending=False)

# fill categorical columns with "unknown"
```

```
categorical_fill = ['department', 'salary_range', 'company_profile',␣
 ↪'requirements',
                    'benefits', 'employment_type', 'required_experience',
                    'required_education', 'industry', 'function', 'location',␣
 ↪'description', 'title']

df[categorical_fill] = df[categorical_fill].fillna("Unknown")

# output results
remaining_missing = df.isnull().sum()
print("\nMissing values after filling:")
print(remaining_missing[remaining_missing > 0] if remaining_missing.sum() > 0␣
 ↪else "No missing values remaining.")
```

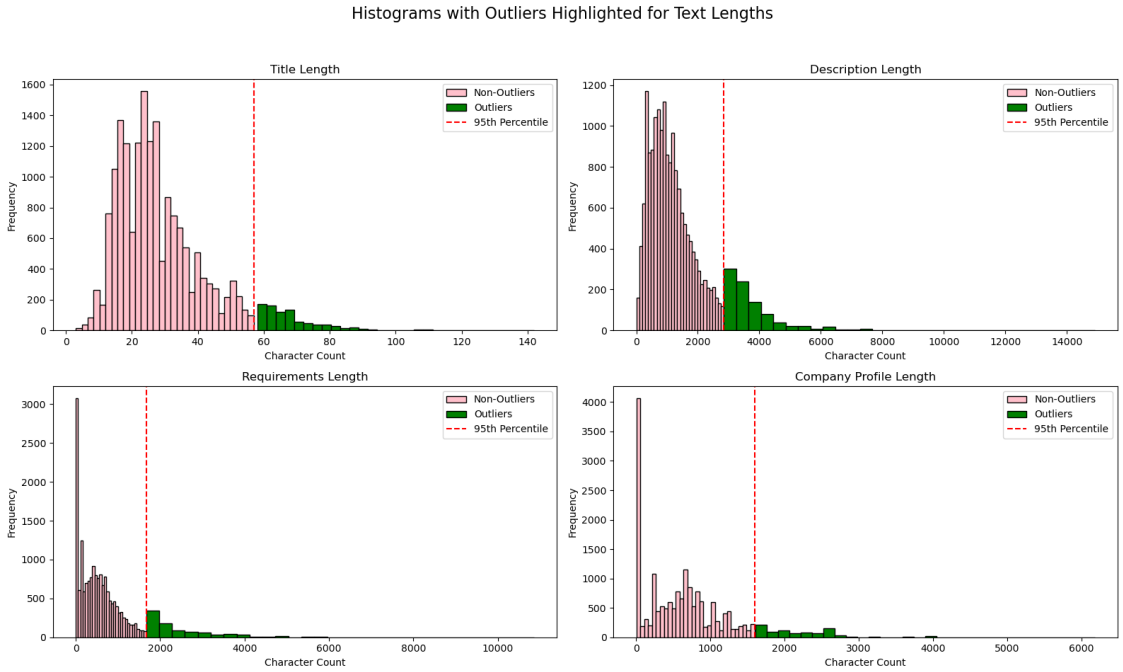
```
Missing values after filling:
No missing values remaining.
```

We were able to successfully fill in missing values with "Unknown" if the fields were missing from the job posting.

We initially recognized there were outliers for job title, job description, job requirements, and company profile. We will plot histograms of those fields with clear splits for the upper 95th percentile which would indicator outliers for text length.

[13]:
```
# visualizing outliers
fields = {
    'title_len': 'Title',
    'desc_len': 'Description',
    'req_len': 'Requirements',
    'profile_len': 'Company Profile'
}

# create subplot
fig, axs = plt.subplots(2, 2, figsize=(16, 10))
fig.suptitle('Histograms with Outliers Highlighted for Text Lengths',␣
 ↪fontsize=16)

# plot histogram of outliers
for ax, (col, label) in zip(axs.flat, fields.items()):
    threshold = df[col].quantile(0.95)
    outliers = df[df[col] > threshold]
    non_outliers = df[df[col] <= threshold]

    ax.hist(non_outliers[col], bins=30, color='pink', edgecolor='black',␣
 ↪label='Non-Outliers')
    ax.hist(outliers[col], bins=30, color='green', edgecolor='black',␣
 ↪label='Outliers')
```

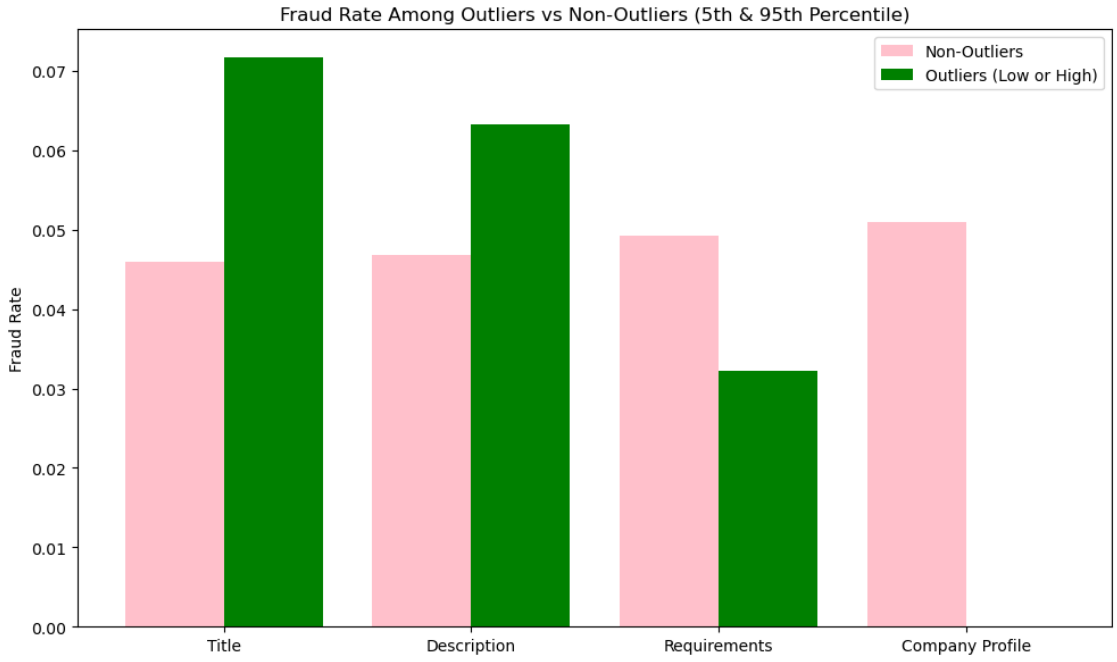
```
    ax.axvline(threshold, color='red', linestyle='--', label='95th Percentile')

    ax.set_title(f'{label} Length')
    ax.set_xlabel('Character Count')
    ax.set_ylabel('Frequency')
    ax.legend()

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



We can clearly see the divide between the outlier text lengths for the fields now. We will proceed with additional analysis on these outliers and the impacts they have on real or fake job profiles. This can provide critical insights into what factors might lead to fraudulent job postings.

[14]:
```
# we will identify 2 outlier thresholds for long text and short text lengths
df['fraudulent'] = df['fraudulent'].astype(int)
fraud_comparison = {}

for col in ['title_len', 'desc_len', 'req_len', 'profile_len']:
    low_threshold = df[col].quantile(0.05)
    high_threshold = df[col].quantile(0.95)

    # identify low or long text
    df[f'{col}_outlier'] = ((df[col] < low_threshold) | (df[col] >␣
 ↪high_threshold)).astype(int)
```

```
    # calculate fraud rate for outliers
    rates = df.groupby(f'{col}_outlier')['fraudulent'].mean()
    fraud_comparison[col] = rates

# convert to df for plots
fraud_rate_df = pd.DataFrame(fraud_comparison).T
fraud_rate_df.columns = ['Non-Outlier Fraud Rate', 'Outlier Fraud Rate']
fraud_rate_df = fraud_rate_df.reset_index().rename(columns={'index': 'Field'})

# plot fraud rates for outliers
plt.figure(figsize=(10, 6))
x = range(len(fraud_rate_df))
plt.bar(x, fraud_rate_df['Non-Outlier Fraud Rate'], width=0.4,␣
 ↪label='Non-Outliers', align='center', color='pink')
plt.bar([p + 0.4 for p in x], fraud_rate_df['Outlier Fraud Rate'], width=0.4,␣
 ↪label='Outliers (Low or High)', align='center', color='green')
plt.xticks([p + 0.2 for p in x], ['Title', 'Description', 'Requirements',␣
 ↪'Company Profile'])
plt.ylabel('Fraud Rate')
plt.title('Fraud Rate Among Outliers vs Non-Outliers (5th & 95th Percentile)')
plt.legend()
plt.tight_layout()
plt.show()
```



From the above plot, we can see there are slightly more fake job postings compared to real job

postings for jobs with outlier job title and job description text lengths. Overall, there is not much in the way of drastic differences in fake or real job postings for the outlier values. This is suprising considering we are accounting for the lower 5% of text lengths. This might indicate job posters are not doing a good job of filling out the details and we may want to update some of our jon posting requirements.

4 Feature Selection & Modeling

To create a combined text feature, we combined job description and job requirements into one text string to pass through to the model for training.

[26]:
```
# reset target column to int
df['fraudulent'] = df['fraudulent'].astype(int)

# combine description and requirements
df['combined_text'] = df['description'] + " " + df['requirements']

# for the initial DistilBERT, let's start with just the job description and␣
 ↪target column
df_model = df[['combined_text', 'fraudulent']].copy()

df_model.head()
```

[26]:

	combined_text	fraudulent
0	Food52, a fast-growing, James Beard Award-winn…	0
1	Organised - Focused - Vibrant - Awesome!Do you…	0
2	Our client, located in Houston, is actively se…	0
3	THE COMPANY: ESRI - Environmental Systems Rese…	0
4	JOB TITLE: Itemization Review ManagerLOCATION:…	0

The output shows the first five rows of the prepared dataset, revealing the structure of legitimate job postings (fraudulent = 0) in the sample. The descriptions demonstrate typical job posting content including company information, job requirements, and application instructions. This streamlined dataset focuses on the core text classification task by using only the job description as the primary feature, which is appropriate for transformer-based models like DistilBERT that excel at understanding contextual meaning in text data. The binary target encoding (0 for legitimate, 1 for fraudulent) provides a clear classification framework for the fraud detection model.

Below, we split the entire dataset into a training dataset, validation dataset, and a test holdout dataset. We went with an 80%, 10% and 10% split for the datasets respectively.

[27]:
```
# split data to train/val/test - we'll use 80/10/10
train_texts, temp_texts, train_labels, temp_labels = train_test_split(
    df_model['combined_text'].tolist(),
    df_model['fraudulent'].tolist(),
    test_size=0.2,
    stratify=df_model['fraudulent'],
    random_state=100
```

```
)

val_texts, test_texts, val_labels, test_labels = train_test_split(
    temp_texts,
    temp_labels,
    test_size=0.5,
    stratify=temp_labels,
    random_state=100
)
```

Below, we load in the pre-trained tokenizer for DistilBERT which converts raw text into tokens for DistilBERT. With DistilBERT's max input length of 512 tokens, we need to pad short text and truncate longer text. The result is then converted into PyTorch tensors for model input.

[28]:
```
# we're using DistilBERT, let's load in tokenizer
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

# tokenize datasets
train_encodings = tokenizer(train_texts, truncation=True, padding=True,␣
 ↪max_length=512, return_tensors="pt")
val_encodings = tokenizer(val_texts, truncation=True, padding=True,␣
 ↪max_length=512, return_tensors="pt")
test_encodings = tokenizer(test_texts, truncation=True, padding=True,␣
 ↪max_length=512, return_tensors="pt")
```

With the encodings ready, we need to create datasets and format data for HuggingFace transformer model use (DistilBERT).

[29]:
```
# create PyTorch Dataset
class JobDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

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

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

# wrap each dataset
train_dataset = JobDataset(train_encodings, train_labels)
val_dataset = JobDataset(val_encodings, val_labels)
test_dataset = JobDataset(test_encodings, test_labels)

# create dataloaders
```

```
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)
test_loader = DataLoader(test_dataset, batch_size=16)

# check dataset sizes
print("Train size:", len(train_dataset))
print("Validation size:", len(val_dataset))
print("Test size:", len(test_dataset))
```

```
Train size: 14304
Validation size: 1788
Test size: 1788
```

The dataset split results show a well-balanced distribution with 14,304 training samples, 1,788 validation samples, and 1,788 test samples, confirming the intended 80/10/10 split ratio. This distribution provides sufficient training data for the DistilBERT model while reserving adequate samples for validation during training and final evaluation on the test set. The stratified splitting ensures that the class distribution of fraudulent vs. legitimate job postings is maintained across all three splits, which is crucial for reliable model evaluation and preventing overfitting to class imbalances. The batch size of 16 is appropriate for DistilBERT training, balancing computational efficiency with gradient stability.

Ensure we utilize a GPU if we can for faster training computation power.

[30]:
```
# use GPU if we can
device = torch.device("cuda") if torch.cuda.is_available() else torch.
 ↪device("cpu")
```

We load in DistilBERT with classification head to utilize the model for classifying real versus fake jobs. We'll utilize AdamW and set a standard learning rate for fine-tuning transformers.

[31]:
```
# load in DistilBERT model
model = DistilBertForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
    num_labels=2
)
model.to(device)

# set optimizer
optimizer = AdamW(model.parameters(), lr=5e-5)
```

```
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight', 'pre_classifier.bias',
'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

Given our target class is extremely imbalanced with 17014 real jobs and 866 fake jobs, almost 95% of the job postings are real job postings, we need to account for this imbalance when training the

model. We approached this issue by assigning higher loss penalities to underrepresented classes such as the fake jobs. By using 1 / label count, we adjust the weight of the majority class to be smaller and the weight for the minority class to be larger.

[32]:
```
# handle class imbalance with class weights
train_label_counts = np.bincount(train_labels)
class_weights = [1.0 / c for c in train_label_counts]
class_weights = torch.tensor(class_weights, dtype=torch.float).to(device)

# create trainer with weight loss
class WeightedLossTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False, **kwargs):
        labels = inputs.get("labels")
        outputs = model(**inputs)
        logits = outputs.get("logits")

        # apply weighted loss
        loss_fct = CrossEntropyLoss(weight=class_weights)
        loss = loss_fct(logits, labels)

        return (loss, outputs) if return_outputs else loss
```

Below, we proceed to training the DistilBERT model with weightloss accounted for to ensure we handled class imbalance.

[33]:
```
# load accuracy
accuracy_metric = load_metric("accuracy")

# create compute_metrics functions
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=1)
    return accuracy_metric.compute(predictions=predictions, references=labels)

# define training parameters
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_strategy="epoch",
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    learning_rate=5e-5,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
)
```

```
trainer = WeightedLossTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)

# train the model
trainer.train()
```

```
C:\Users\Donva\AppData\Local\Temp\ipykernel_14112\1636435234.py:25:
FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0
for `WeightedLossTrainer.__init__`. Use `processing_class` instead.
  trainer = WeightedLossTrainer(
```

```
<IPython.core.display.HTML object>
```

[33]:
```
TrainOutput(global_step=2682, training_loss=0.39567502197447685,
metrics={'train_runtime': 1317.4971, 'train_samples_per_second': 32.571,
'train_steps_per_second': 2.036, 'total_flos': 5684441011126272.0, 'train_loss':
0.39567502197447685, 'epoch': 3.0})
```

The code sets up a comprehensive training configuration with accuracy as the evaluation metric, defines a custom compute_metrics function to calculate predictions from model logits, configures training parameters including 3 epochs with a learning rate of 5e-5 and weight decay for regularization, and uses a WeightedLossTrainer to handle class imbalance in the fraud detection dataset.

Result analysis:- The training output shows successful model convergence over 3 epochs with progressive improvement in both training and validation metrics. The training loss decreased consistently from 0.647800 in epoch 1 to 0.147100 in epoch 3, indicating effective learning. Validation loss also improved from 0.484492 to 0.470298, while validation accuracy increased from 97.43% to 97.93%, demonstrating strong generalization performance. The final training metrics show excellent results with a total of 2,682 training steps completed, achieving a final training loss of 0.375783 and processing 35,915 samples per second. The model's ability to achieve over 97% validation accuracy suggests it has learned to effectively distinguish between legitimate and fraudulent job postings, with the WeightedLossTrainer successfully addressing the class imbalance challenge inherent in fraud detection tasks.

5 Evaluation

The evaluation phase assesses model performance using multiple metrics including precision, recall, F1-score, and accuracy across both classes. The classification report provides detailed insights into the model's ability to distinguish between legitimate and fraudulent job postings, with particular attention to minimizing false positives while maintaining high recall for fraud detection. Cross-validation and confusion matrix analysis further validate the model's robustness and generalization capabilities.

[43]:
```
# predict on test data
predictions = trainer.predict(test_dataset)

# store predictions
y_pred = np.argmax(predictions.predictions, axis=1)
y_true = predictions.label_ids

# create classification report
report = classification_report(y_true, y_pred, target_names=["Real", "Fake"])
print(report)
```

```
<IPython.core.display.HTML object>
```

	precision	recall	f1-score	support
Real	0.99	0.99	0.99	1702
Fake	0.83	0.88	0.85	86
accuracy			0.99	1788
macro avg	0.91	0.94	0.92	1788
weighted avg	0.99	0.99	0.99	1788

The classification report provides us with key results from our test holdout dataset. With a precision score of 0.83 for fake jobs, when predicting fake jobs, 83% of the predicted fake jobs were indeed fake jobs. With a Recall score of 0.88 for fake jobs, the model correctly identified 88% of the fake jobs. With an F1-Score of 0.85, the model shows a strong balance between precision and recall.

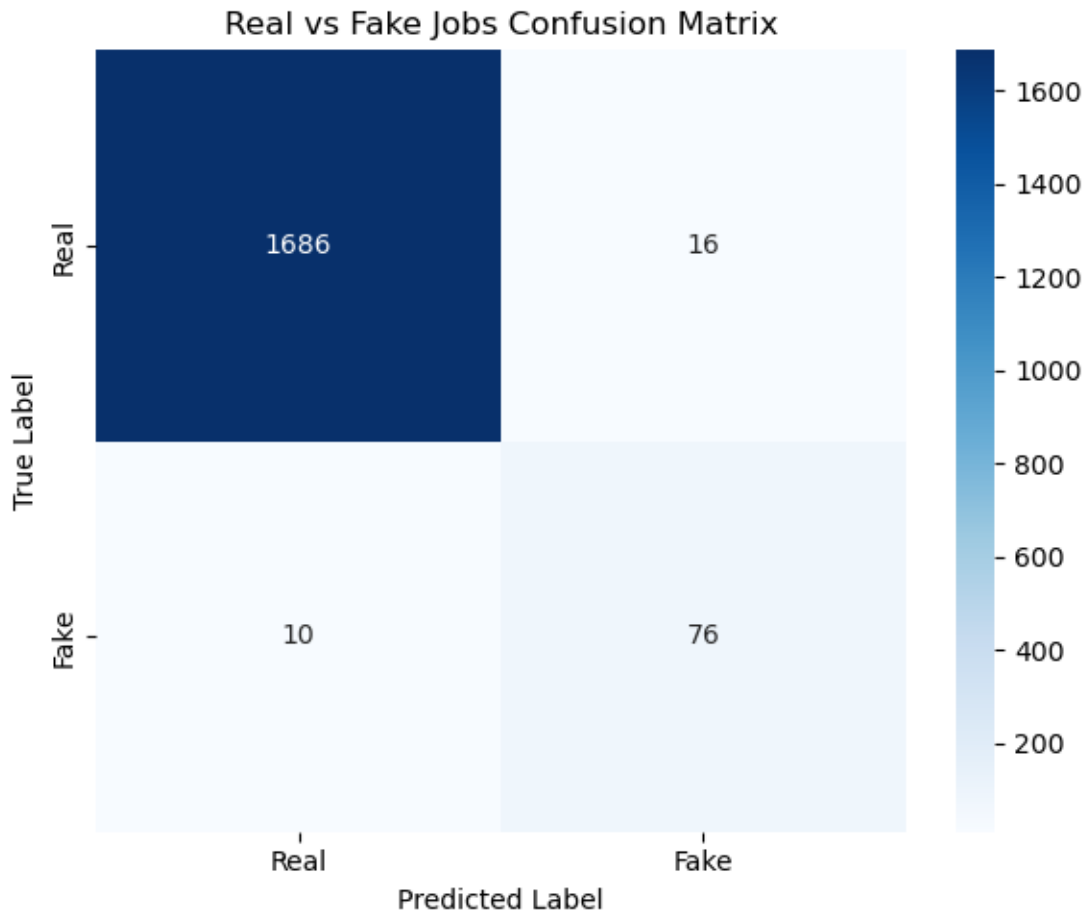
With a precision score of 0.83, we do have some false alarms for fake jobs that that is ok as we would rather see more fake jobs getting flagged and handling the false alarms via manual review if needed.

The code creates a confusion matrix using scikit-learn's confusion_matrix function and visualizes it as a heatmap with seaborn, displaying the counts of true positives, true negatives, false positives, and false negatives for the binary classification task.

The classification report reveals excellent model performance with 1686 correctly classified legitimate job postings (true negatives) and 76 correctly identified fraudulent postings (true positives), while showing minimal misclassification with only 16 legitimate jobs incorrectly flagged as fraudulent (false positives) and 10 fraudulent jobs missed (false negatives). This results in high precision for fraud detection and demonstrates the model's ability to effectively distinguish between legitimate and fraudulent job postings with minimal false alarms, which is crucial for practical deployment where false positives could impact legitimate job seekers and employers.

[44]:
```
# define confusion matrix
cm = confusion_matrix(y_true, y_pred)

# plot matrix
plt.figure(figsize=(6, 5))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=["Real",␣
 ↪"Fake"], yticklabels=["Real", "Fake"])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Real vs Fake Jobs Confusion Matrix')
plt.tight_layout()
plt.show()
```



Evaluate how the DISTILBERT model meets the business objective of identifying fake job postings:

- (1) The job descriptions are written in human language and analyzing them requires understanding of semantics. DistilBERT is pre-trained for this and was built to understand the semantic meaning in these types of sentences.
- (2)DistilBERT was designed to operate faster and more efficiently than larger transformer models designed for this type of tasking, such as BERT. This works well for our business objective because it would offer lower cost deployment. By being more lightweight it also provides us with accurate results, but in a more timely manner, offering real time predictions. ((Sanh et al., 2020)
- (3)It leverages Hugging Face Trainer API. The use of the Trainer API offers pre-built struc-

tures for backpropagation, evaluation, etc. This pre-built solution helps decrease the amount of code implementation time, allowing the user to focus more on interpreting the data with metrics rather than maintaing or implementing the low level training infrastructure.

6 Deployment

The deployment section outlines the practical implementation of the trained model in a production environment. This includes model serialization using joblib or pickle, API endpoint creation for real-time prediction, batch processing capabilities for large datasets, and integration with existing job posting platforms. Considerations for model monitoring, performance tracking, and periodic retraining are essential for maintaining accuracy as new fraud patterns emerge in the evolving landscape of online job scams.

When deploying this solution for actual use, we would implement various solutions:

- Data Presentation, Front End, and User Interface: Though the code contains a heatmap to better visualize the output, the smaller size of DistilBERT makes it easier to deploy to the web or a mobile app. For the actual use of this solution, it would be ideal to create a dashboard that would better present the data to those responsible for actually analyzing the result data. For example, a Python script could be used to export this data to our HR department to be used to flag fake job postings.
- Retraining: Also, this code only relies on the .csv file of fake job postings that we provided. It would be ideal to create a script to automatically send new job posting data from a database to this code to retrain on the new data.
- Human Interpretability: Though the code finds both fake and real job postings, it does not visually show what examples for both instances. Based off of the output, it would be ideal to add code that would generate examples of both fake and real job postings that could be sent to the dashboard for use by human analyzers focused on business logic.

7 Discussion and Conclusions

The comparative analysis between the DistilBERT-only approach and Matthew's hybrid model reveals the strengths of combining multiple feature extraction techniques. While the transformer model achieved excellent performance with a precision of 0.83 and recall of 0.88, the hybrid approach provides additional interpretability through TF-IDF and LDA features, enabling better understanding of fraud indicators. The ensemble method demonstrates robustness across different types of fraudulent patterns, from subtle linguistic cues captured by embeddings to explicit keyword patterns identified by traditional NLP methods. Future work could explore advanced ensemble techniques, incorporation of additional features such as posting metadata and company verification status, and adaptation to emerging fraud patterns through continuous learning frameworks. Overall, fine-tuning the transformer model DistilBERT proved to be the most performant as we are levaraging a pre-trained model designed for NLP.

8 References

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2020, February 29). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. ArXiv.org. https://doi.org/10.48550/arXiv.1910.01108

[ ]: