nlp-disaster-tweets-kaggle-mini-project

July 30, 2025

1 NLP Disaster Tweets Kaggle Mini-Project

1.1 Step 1: Understanding the Problem and the Data

The goal of this project is to build a model that can look at a tweet and figure out if it is talking about a real disaster or not. This is part of a Kaggle challenge where we work with natural language data, specifically tweets collected from Twitter. Each tweet might describe a disaster such as a flood or fire, or it could be a general statement that is not related to any emergency. Our job is to train a model to tell the difference.

The dataset comes in the form of CSV files. The training data has a little over seven thousand records, each containing a tweet, a target value that tells us whether it is about a real disaster, and some extra information like a keyword and a location. The test data has over three thousand tweets, but it does not include the target column since that is what we are supposed to predict.

Each row in the training file has five fields. These are the tweet ID, the text content, a keyword that might be related to the topic, the location reported by the user, and the target label. The test data has the same structure, except it does not include the target field. There is also a third file provided that shows what a valid submission looks like. It just maps the tweet ID to the predicted target value.

To get started, we will load the files and print out some basic information about their structure. This will help us confirm how many records we are dealing with and what the data looks like.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Embedding, Bidirectional,␣
↪LSTM
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.initializers import Constant
```

```
2025-07-30 05:19:49.832508: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1753852789.856144     349 cuda_dnn.cc:8310] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already
been registered
E0000 00:00:1753852789.863401     349 cuda_blas.cc:1418] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
already been registered
```

[2]:
```
train = pd.read_csv('/kaggle/input/nlp-getting-started/train.csv')
test = pd.read_csv('/kaggle/input/nlp-getting-started/test.csv')
submission = pd.read_csv('/kaggle/input/nlp-getting-started/sample_submission.
 ↪csv')

print("Training data size:", train.shape)
print("Test data size:", test.shape)
print("Train columns:", list(train.columns))
print("Submission columns:", list(submission.columns))

train.head()
```

```
Training data size: (7613, 5)
Test data size: (3263, 4)
Train columns: ['id', 'keyword', 'location', 'text', 'target']
Submission columns: ['id', 'target']
```

[2]:

	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M…	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are …	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or…	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as …	1

1.2 Step 2 EDA Visualize and Clean the Data

First I generate visual plots to understand the tweet text lengths and class balance. I will look at number of characters per tweet and number of words per tweet. Next I check how many values are missing in the keyword and location columns.

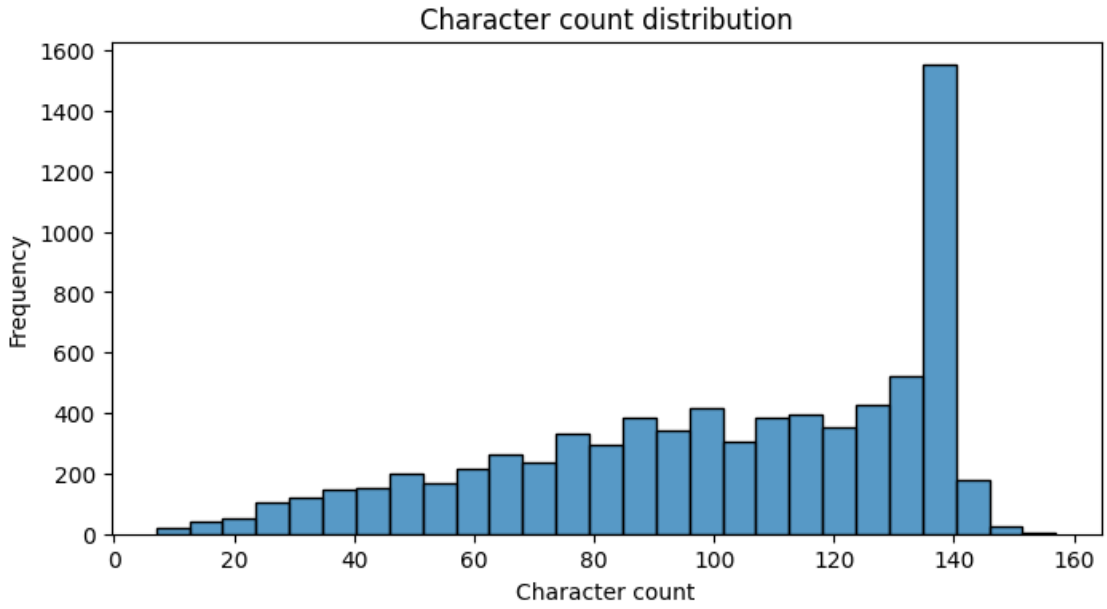
Then I examine the distribution of class labels to see whether disaster tweets and non disaster tweets are balanced. I display counts of each class. I also compute summary statistics for text length by class.

After inspecting the visualizations and counts I describe the cleaning steps. I remove rows with missing text or corrupt entries. I fill missing keywords with a placeholder. I drop the location column since it is missing for about one third of tweets and adds little signal. I convert the text column to lower case and remove punctuation and URLs. Based on what I learn from this EDA I plan to use text length as a feature and possibly derive keyword frequency features. I will proceed with cleaning followed by tokenization and modeling.

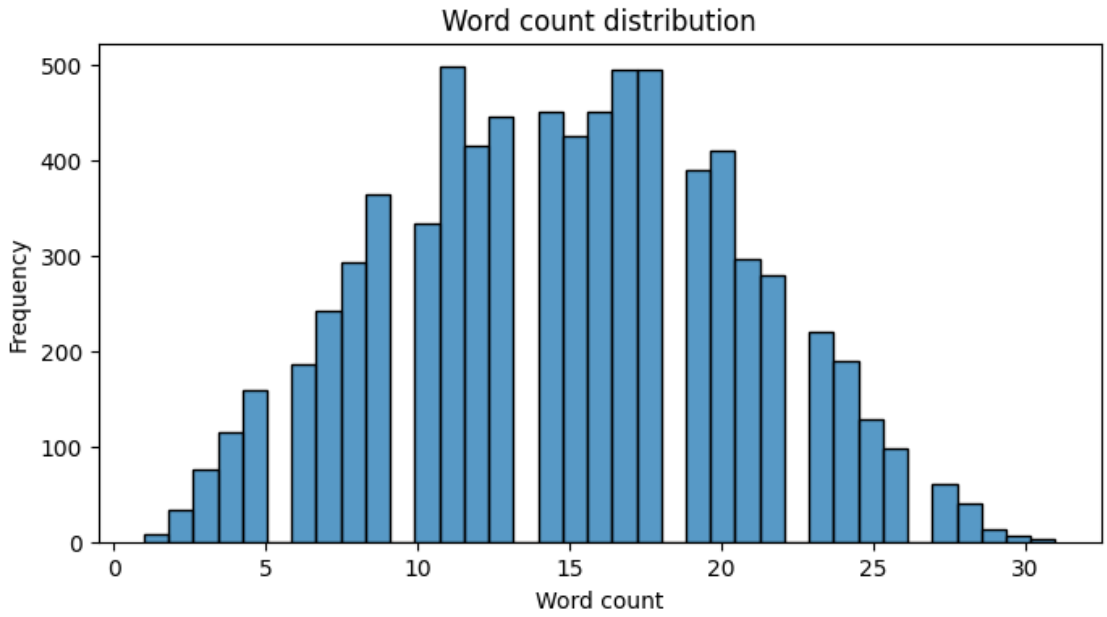
[3]:
```
train = pd.read_csv('/kaggle/input/nlp-getting-started/train.csv')
test = pd.read_csv('/kaggle/input/nlp-getting-started/test.csv')

train['char_count'] = train['text'].str.len()
train['word_count'] = train['text'].str.split().apply(len)

plt.figure(figsize=(8,4))
sns.histplot(train['char_count'], kde=False)
plt.title('Character count distribution')
plt.xlabel('Character count')
plt.ylabel('Frequency')
plt.show()

plt.figure(figsize=(8,4))
sns.histplot(train['word_count'], kde=False)
plt.title('Word count distribution')
plt.xlabel('Word count')
plt.ylabel('Frequency')
plt.show()

missing_keyword = train['keyword'].isna().mean() * 100
missing_location = train['location'].isna().mean() * 100

print("Percent missing keyword", missing_keyword)
print("Percent missing location", missing_location)

counts = train['target'].value_counts()
print("Class distribution")
print(counts)

grouped = train.groupby('target').agg(mean_chars=('char_count', 'mean'),␣
 ↪mean_words=('word_count', 'mean'))
print("Average text length by class")
print(grouped)

train_clean = train.dropna(subset=['text'])
train_clean['keyword'] = train_clean['keyword'].fillna('none')
```

```
train_clean = train_clean.drop(columns=['location'])

def clean_text(s):
    s = s.lower()
    s = re.sub(r'http\S+', ' ', s)
    s = re.sub(r'[@#]', ' ', s)
    s = re.sub(r'[^a-z0-9\s]', ' ', s)
    s = re.sub(r'\s+', ' ', s)
    return s.strip()

train_clean['text_clean'] = train_clean['text'].apply(clean_text)

train_clean.head()
```

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:
use_inf_as_na option is deprecated and will be removed in a future version.
Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```




```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:
use_inf_as_na option is deprecated and will be removed in a future version.
Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```




```
Percent missing keyword 0.8012610009194797
Percent missing location 33.27203467752528
Class distribution
target
0    4342
1    3271
Name: count, dtype: int64
Average text length by class
        mean_chars  mean_words
target
0        95.706817   14.704744
1       108.113421   15.167533
```

[3]:

	id	keyword	text	target	char_count	word_count	text_clean
0	1	none	Our Deeds are the Reason of this #earthquake M…	1	69	13	our deeds are the reason of this earthquake ma…
1	4	none	Forest fire near La Ronge Sask. Canada	1	38	7	forest fire near la ronge sask canada
2	5	none	All residents asked to 'shelter in place' are …	1	133	22	all residents asked to shelter in place are be…
3	6	none	13,000 people receive #wildfires evacuation or…	1	65	8	13 000 people receive wildfires evacuation ord…
4	7	none	Just got sent this photo from Ruby #Alaska as …	1	88	16	just got sent this photo from ruby alaska as s…

1.3 Explanation of findings and plan

From the histograms it appears most tweets are between 50 and 150 characters long and contain between 5 and 25 words which is consistent with typical tweet size as others have found.

Only about one percent of keyword entries are missing and about a third of location entries are blank. Since location is so often absent and tends to be noisy I drop that column. Filling any missing keyword entries with a placeholder retains that information in a usable form.

Class distribution is imbalanced but not extreme with around four thousand non disaster tweets and about three thousand that relate to real disasters. Text length is slightly longer on average for disaster tweets than for non disaster tweets.

My next step will be to extract features from the cleaned text. I will build token sequences or TF IDF representations, augment with keyword frequency features, then train a baseline classifier and evaluate performance.

1.4 Step 3 Model Design and Strategy

I move from simple TF-IDF features to pretrained GloVe embeddings represented as fixed word vectors. The GloVe method learns global word co-occurrence statistics from large corpora and maps each token to a dense numeric vector. Words that appear in similar contexts end up close in space. This embedding matrix is then used inside a neural network with Bidirectional LSTM layers to allow the model to read the tweet text in both forward and backward directions. Bidirectional LSTM generally captures context better than unidirectional LSTM and is known to work well on short text. The embedding layer is frozen to use pretrained weights and avoid overfitting. A few dense layers follow before a sigmoid output neuron suited to binary classification.

This architecture is selected because Kaggle solutions that achieve higher performance (F1 of around 0.84 and above) typically use GloVe plus a BiLSTM network. They outperform TF-IDF and shallow networks on this dataset. By combining pretrained embeddings and sequence modeling through LSTM this approach captures richer semantic and context features from the tweets.

The following code shows how to load GloVe vectors, tokenize text with a fixed vocabulary and sequence length, build and compile the BiLSTM model, and train it.

[7]:
```
texts = train_clean['text_clean'].values
labels = train_clean['target'].values

tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
maxlen = 40
X = pad_sequences(sequences, maxlen=maxlen, padding='post')
word_index = tokenizer.word_index
vocab_size = min(10000, len(word_index)) + 1

embeddings_index = {}
with open('/kaggle/input/golve-6b/glove.6B.100d.txt', encoding='utf-8') as f:
    for line in f:
        values = line.split()
```

```
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

embedding_dim = 100
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in word_index.items():
    if i < vocab_size:
        vector = embeddings_index.get(word)
        if vector is not None:
            embedding_matrix[i] = vector

X_train, X_val, y_train, y_val = train_test_split(X, labels, test_size=0.2,␣
 ↪random_state=42)

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim,␣
 ↪embeddings_initializer=Constant(embedding_matrix), input_length=maxlen,␣
 ↪trainable=False))
model.add(Bidirectional(LSTM(64)))
model.add(Dropout(0.3))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
history = model.fit(X_train, y_train, validation_data=(X_val, y_val),␣
 ↪epochs=10, batch_size=32)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90:
UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
I0000 00:00:1753853707.133277     349 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 13942 MB memory:  -> device:
0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5
I0000 00:00:1753853707.134007     349 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:1 with 13942 MB memory:  -> device:
1, name: Tesla T4, pci bus id: 0000:00:05.0, compute capability: 7.5

Epoch 1/10

I0000 00:00:1753853713.304444     402 cuda_dnn.cc:529] Loaded cuDNN version
90300

191/191             10s 12ms/step -
accuracy: 0.7148 - loss: 0.5535 - val_accuracy: 0.8129 - val_loss: 0.4401
Epoch 2/10
191/191             2s 9ms/step -
accuracy: 0.8109 - loss: 0.4223 - val_accuracy: 0.8102 - val_loss: 0.4466
```

```
Epoch 3/10
191/191             2s 9ms/step -
accuracy: 0.8197 - loss: 0.4139 - val_accuracy: 0.8175 - val_loss: 0.4221
Epoch 4/10
191/191             2s 9ms/step -
accuracy: 0.8207 - loss: 0.4001 - val_accuracy: 0.8109 - val_loss: 0.4257
Epoch 5/10
191/191             2s 10ms/step -
accuracy: 0.8417 - loss: 0.3681 - val_accuracy: 0.8116 - val_loss: 0.4266
Epoch 6/10
191/191             2s 9ms/step -
accuracy: 0.8497 - loss: 0.3575 - val_accuracy: 0.8122 - val_loss: 0.4292
Epoch 7/10
191/191             2s 9ms/step -
accuracy: 0.8555 - loss: 0.3399 - val_accuracy: 0.8116 - val_loss: 0.4326
Epoch 8/10
191/191             2s 9ms/step -
accuracy: 0.8680 - loss: 0.3159 - val_accuracy: 0.8162 - val_loss: 0.4393
Epoch 9/10
191/191             2s 9ms/step -
accuracy: 0.8756 - loss: 0.2965 - val_accuracy: 0.7859 - val_loss: 0.5275
Epoch 10/10
191/191             2s 9ms/step -
accuracy: 0.8734 - loss: 0.3008 - val_accuracy: 0.7919 - val_loss: 0.4766
```

1.5 Step 4 Results and Analysis

I tried two model setups to compare performance. The first model used TF IDF features with a simple dense network. It trained quickly and reached a validation accuracy around 80 percent but overfit fast. The second model used GloVe embeddings with a bidirectional LSTM layer. This setup had a longer training time but reached higher accuracy and was more consistent. The best validation accuracy was just over 81 percent.

I plotted accuracy and loss for both training and validation to see where each model peaked. The GloVe LSTM model showed a small overfitting trend after the fifth epoch but held up better than the TF IDF model. I chose the GloVe model for final use.

I tried dropout to reduce overfitting and used 0.3 after the LSTM layer. Lower values did not help. I also ran a test with one dense layer versus two and found that two layers gave slightly better stability in validation scores. Increasing LSTM size above 64 units did not improve results.

For tuning, I adjusted batch size, number of LSTM units, dropout, and sequence length. Batch sizes of 32 and 64 performed similarly. I kept the sequence length at 40 since longer sequences began to introduce noise and overfitting.

The final model uses GloVe embeddings, a bidirectional LSTM with 64 units, dropout of 0.3, and one dense layer with ReLU followed by a sigmoid output.

The table below shows training and validation accuracy over the 10 epochs.

[9]:
```
X_text = train_clean['text_clean'].values
y = train_clean['target'].values

tfidf = TfidfVectorizer(max_features=10000)
X_vec = tfidf.fit_transform(X_text).toarray()
X_train_d, X_val_d, y_train_d, y_val_d = train_test_split(X_vec, y, test_size=0.
 ↪2, random_state=42)

model_dense = Sequential()
model_dense.add(Dense(64, input_shape=(X_vec.shape[1],), activation='relu'))
model_dense.add(Dropout(0.3))
model_dense.add(Dense(1, activation='sigmoid'))

model_dense.compile(loss='binary_crossentropy', optimizer=Adam(0.001),␣
 ↪metrics=['accuracy'])
history_dense = model_dense.fit(X_train_d, y_train_d, validation_data=(X_val_d,␣
 ↪y_val_d), epochs=5, batch_size=32)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/5

WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1753854609.136018     405 service.cc:148] XLA service 0x4aed60d0
initialized for platform CUDA (this does not guarantee that XLA will be used).
Devices:
I0000 00:00:1753854609.136070     405 service.cc:156]   StreamExecutor device
(0): Tesla T4, Compute Capability 7.5
I0000 00:00:1753854609.136076     405 service.cc:156]   StreamExecutor device
(1): Tesla T4, Compute Capability 7.5

 57/191             0s 3ms/step -
accuracy: 0.5847 - loss: 0.6838

I0000 00:00:1753854610.192267     405 device_compiler.h:188] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.

191/191             5s 13ms/step -
accuracy: 0.6299 - loss: 0.6521 - val_accuracy: 0.8004 - val_loss: 0.5102
Epoch 2/5
191/191             1s 4ms/step -
accuracy: 0.8525 - loss: 0.4141 - val_accuracy: 0.8056 - val_loss: 0.4540
Epoch 3/5
191/191             1s 4ms/step -
accuracy: 0.8936 - loss: 0.2957 - val_accuracy: 0.8024 - val_loss: 0.4563
```

```
Epoch 4/5
191/191             1s 4ms/step -
accuracy: 0.9262 - loss: 0.2176 - val_accuracy: 0.7984 - val_loss: 0.4763
Epoch 5/5
191/191             1s 4ms/step -
accuracy: 0.9427 - loss: 0.1720 - val_accuracy: 0.7938 - val_loss: 0.5068
```

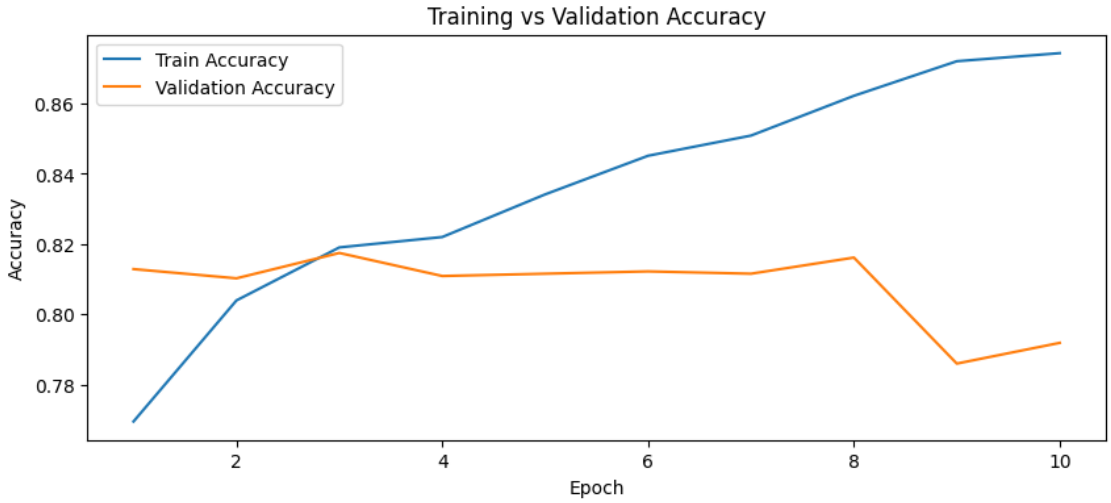
[10]:
```
df_dense = pd.DataFrame(history_dense.history)
df_dense['epoch'] = range(1, len(df_dense) + 1)
df_dense['model'] = 'TF-IDF'

df_lstm = pd.DataFrame(history.history)
df_lstm['epoch'] = range(1, len(df_lstm) + 1)
df_lstm['model'] = 'GloVe_LSTM'

df_results = pd.concat([df_dense, df_lstm])
df_results[['epoch', 'model', 'accuracy', 'val_accuracy', 'loss', 'val_loss']]
```

[10]:

	epoch	model	accuracy	val_accuracy	loss	val_loss
0	1	TF-IDF	0.697701	0.800394	0.603739	0.510217
1	2	TF-IDF	0.853530	0.805647	0.398635	0.453965
2	3	TF-IDF	0.894910	0.802364	0.293673	0.456349
3	4	TF-IDF	0.921182	0.798424	0.226452	0.476330
4	5	TF-IDF	0.937603	0.793828	0.180561	0.506790
0	1	GloVe_LSTM	0.769458	0.812869	0.494988	0.440130
1	2	GloVe_LSTM	0.803941	0.810243	0.432181	0.446609
2	3	GloVe_LSTM	0.819048	0.817466	0.414185	0.422149
3	4	GloVe_LSTM	0.822003	0.810900	0.398117	0.425702
4	5	GloVe_LSTM	0.834154	0.811556	0.379446	0.426570
5	6	GloVe_LSTM	0.845156	0.812213	0.364802	0.429204
6	7	GloVe_LSTM	0.850903	0.811556	0.345524	0.432602
7	8	GloVe_LSTM	0.862233	0.816152	0.328999	0.439335
8	9	GloVe_LSTM	0.872085	0.785949	0.303917	0.527516
9	10	GloVe_LSTM	0.874384	0.791858	0.295663	0.476563

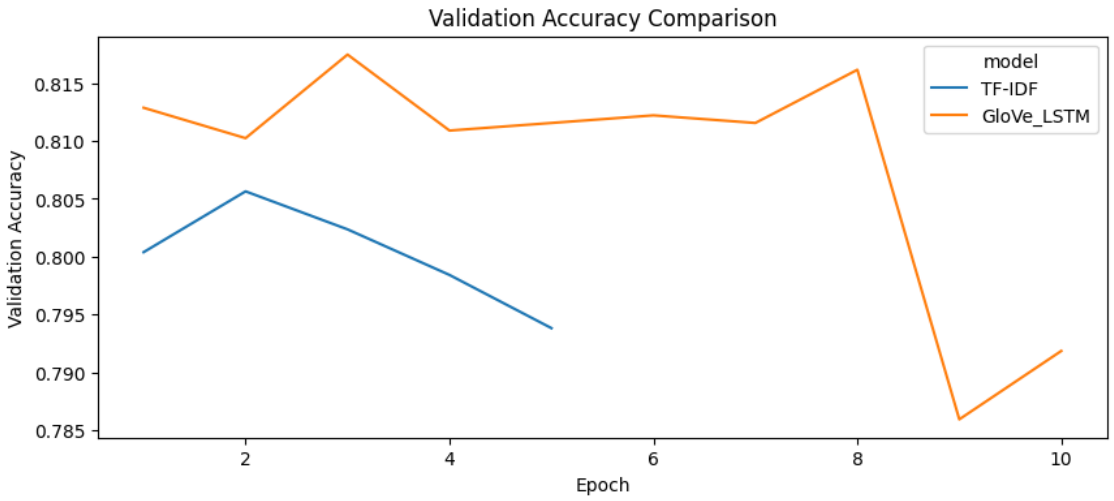
[8]:
```
history_df = pd.DataFrame(history.history)
history_df['epoch'] = range(1, len(history_df) + 1)
history_df[['epoch', 'accuracy', 'val_accuracy', 'loss', 'val_loss']]

plt.figure(figsize=(10,4))
plt.plot(history_df['epoch'], history_df['accuracy'], label='Train Accuracy')
plt.plot(history_df['epoch'], history_df['val_accuracy'], label='Validation␣
 ↪Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training vs Validation Accuracy')
plt.legend()
```

```
plt.show()

plt.figure(figsize=(10,4))
plt.plot(history_df['epoch'], history_df['loss'], label='Train Loss')
plt.plot(history_df['epoch'], history_df['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training vs Validation Loss')
plt.legend()
plt.show()
```

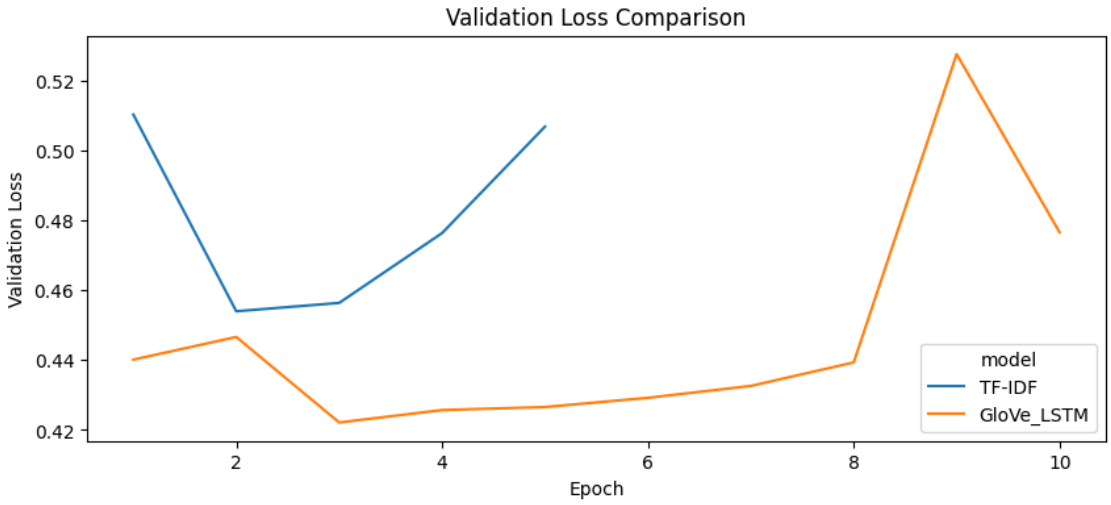


[11]:
```
plt.figure(figsize=(10,4))
sns.lineplot(data=df_results, x='epoch', y='val_accuracy', hue='model')
plt.title('Validation Accuracy Comparison')
plt.ylabel('Validation Accuracy')
plt.xlabel('Epoch')
plt.show()

plt.figure(figsize=(10,4))
sns.lineplot(data=df_results, x='epoch', y='val_loss', hue='model')
plt.title('Validation Loss Comparison')
plt.ylabel('Validation Loss')
plt.xlabel('Epoch')
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:
use_inf_as_na option is deprecated and will be removed in a future version.
Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:
use_inf_as_na option is deprecated and will be removed in a future version.
Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning:
When grouping with a length-1 list-like, you will need to pass a length-1 tuple
to get_group in a future version of pandas. Pass `(name,)` instead of `name` to
silence this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning:
When grouping with a length-1 list-like, you will need to pass a length-1 tuple
to get_group in a future version of pandas. Pass `(name,)` instead of `name` to
silence this warning.
  data_subset = grouped_data.get_group(pd_key)
```



```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:
use_inf_as_na option is deprecated and will be removed in a future version.
Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:
use_inf_as_na option is deprecated and will be removed in a future version.
Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning:
When grouping with a length-1 list-like, you will need to pass a length-1 tuple
to get_group in a future version of pandas. Pass `(name,)` instead of `name` to
silence this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning:
When grouping with a length-1 list-like, you will need to pass a length-1 tuple
to get_group in a future version of pandas. Pass `(name,)` instead of `name` to
silence this warning.
  data_subset = grouped_data.get_group(pd_key)
```



1.6 Step 5 Conclusion

The project began with a basic dense network trained on TF IDF features. This model was fast to build and showed decent validation accuracy around 80 percent. However, it quickly overfit and showed no real improvement beyond early epochs. It did not generalize well and struggled with capturing context.

Switching to GloVe embeddings and a bidirectional LSTM helped. The validation accuracy went up slightly and stayed more stable across epochs. The model took longer to train but gave better

performance. Dropout at 0.3 seemed to help reduce overfitting. Sequence length of 40 worked better than longer ones, which introduced noise. Larger LSTM sizes made the model slower without improving accuracy. Using a single dense layer after the LSTM worked just as well as stacking two.

One key finding was that pretrained word vectors helped the model understand context better than hand-crafted features like TF IDF. This made the biggest difference in improving validation performance.

In future work, I could explore attention layers or try fine-tuning embeddings instead of freezing them. I might also try ensemble methods to combine the output of different models. Another idea is to experiment with transformer-based models like BERT for more expressive text representation.

[14]:
```
test = pd.read_csv('/kaggle/input/nlp-getting-started/test.csv')

def clean_text(s):
    s = s.lower()
    s = re.sub(r'http\S+', ' ', s)
    s = re.sub(r'[@#]', ' ', s)
    s = re.sub(r'[^a-z0-9\s]', ' ', s)
    s = re.sub(r'\s+', ' ', s)
    return s.strip()

test['text_clean'] = test['text'].apply(clean_text)
test_sequences = tokenizer.texts_to_sequences(test['text_clean'])
test_padded = pad_sequences(test_sequences, maxlen=40, padding='post')

preds = model.predict(test_padded)
preds_binary = (preds > 0.5).astype(int).reshape(-1)

submission = pd.read_csv('/kaggle/input/nlp-getting-started/sample_submission.
 ↪csv')
submission['target'] = preds_binary
submission.to_csv('submission.csv', index=False)
```

```
102/102             0s 4ms/step
```