

HANI NOORBUX SHAIKH

JIGNESH PRATAPJIBHAI VAGHELA

SEHREEN KHAN

DECLAN TREVOR KINTU

# FINAL PROJECT.

---

*ONTARIO LANDLORD AND TENANT TRIBUNAL CHATBOT ASSISTANT.*

GROUP 3

AIDI 2001 | PROFESSOR UZAIR AHMAD

## Table of Contents

<b>INTRODUCTION:</b> .....	2
<b>MODEL DESCRIPTION:</b> .....	2
<b>1) <i>Import Libraries.</i></b> .....	2
<b>2) <i>Data Manipulation.</i></b> .....	3
<b>3) <i>Create Models.</i></b> .....	3
<b>4) <i>Define Functions.</i></b> .....	6
<b>5) <i>Fine-tuning.</i></b> .....	7
<b>6) <i>User Interfaces.</i></b> .....	9
<b>OUR SUGGESTION:</b> .....	10

# INTRODUCTION:

For this project, our team decided to train two models to generate an AI chatbot for the Ontario Landlord and Tenant Tribunal. The two Machine Learning models we decided to adapt for this solution are a Multinomial Naïve Bayes model and a Long Short-Term Memory (LSTM) model.

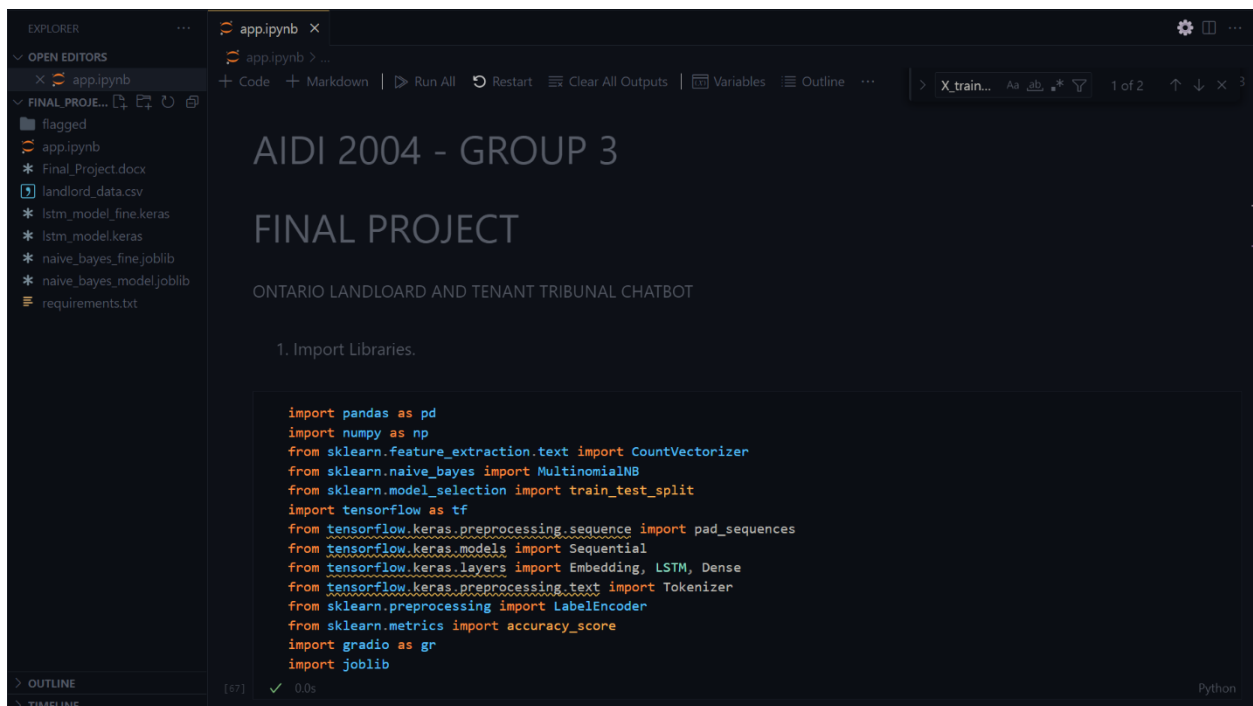
Within this project documentation, we shall describe the process of building both models and give our assessment of both models at the end.

## MODEL DESCRIPTION:

Below are screenshots defining our model-building process.

### 1) *Import Libraries.*

We import the necessary libraries required to manipulate the data and build our ML models.

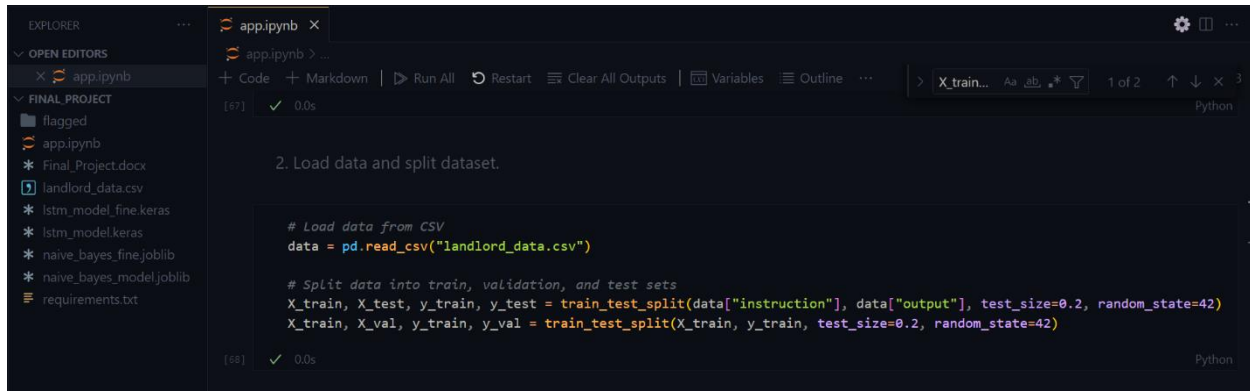


```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
import gradio as gr
import joblib
```

## 2) Data Manipulation.

The data is loaded from a csv file containing various prompts given to the [official Tribunal website](#) navigation and the responses that were shared by the public expert system.

The data is then split into two sets: a training set and a testing set.



The screenshot shows a Jupyter Notebook interface with a dark theme. On the left, the 'EXPLORER' sidebar lists files in the 'FINAL PROJECT' directory, including 'flagged', 'app.ipynb', 'Final\_Project.docx', 'landlord\_data.csv', 'lstm\_model\_fine.keras', 'lstm\_model.keras', 'naive\_bayes\_fine.joblib', 'naive\_bayes\_model.joblib', and 'requirements.txt'. The main editor area displays the code for the second step: '2. Load data and split dataset.' The code includes comments and uses pandas and sklearn to load the CSV file and split the data into training and testing sets.

```
[67] ✓ 0.0s  
  
2. Load data and split dataset.  
  
# Load data from CSV  
data = pd.read_csv("landlord_data.csv")  
  
# Split data into train, validation, and test sets  
X_train, X_test, y_train, y_test = train_test_split(data["instruction"], data["output"], test_size=0.2, random_state=42)  
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)  
  
[68] ✓ 0.0s
```

## 3) Create Models.

The Multinomial Naïve Bayes Model is defined and trained.

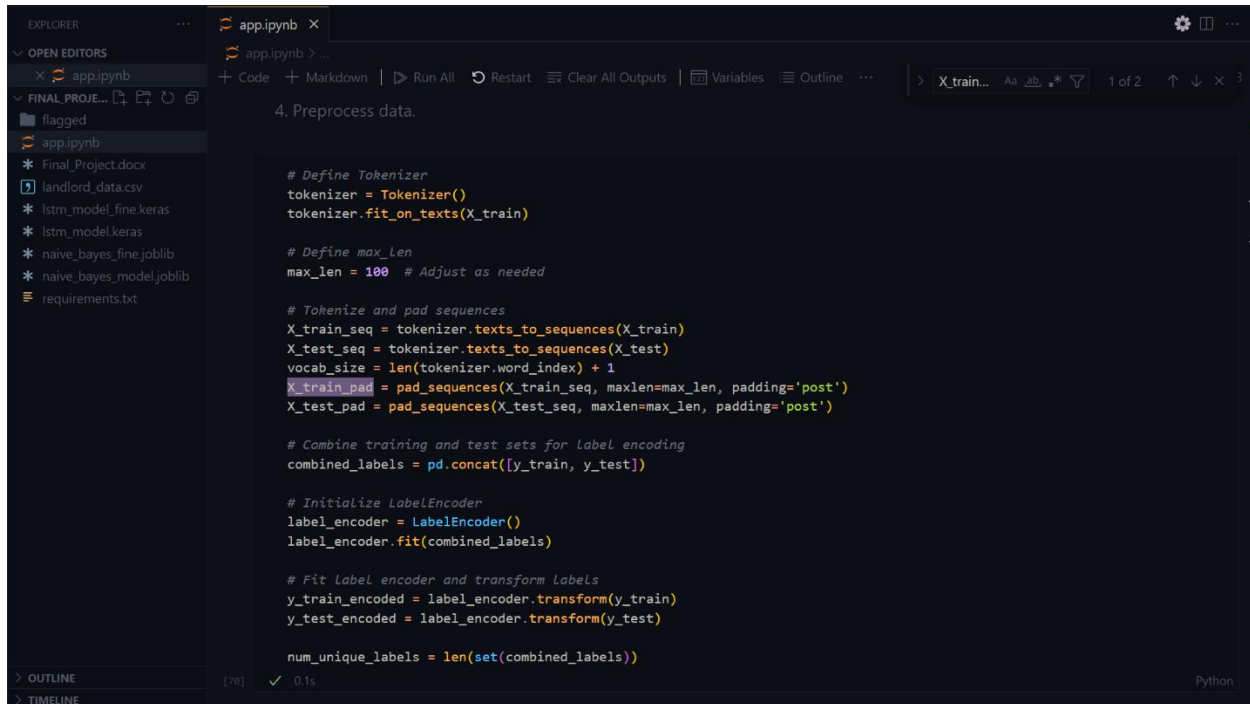


The screenshot shows a Jupyter Notebook interface with a dark theme. The main editor area displays the code for the third step: '3. Define Multinomial Naive Bayes model.' The code includes comments and uses sklearn to define a vectorizer, transform the training data, and train a Multinomial Naive Bayes classifier. The 'OUTLINE' sidebar on the left shows the current cell's content.

```
3. Define Multinomial Naive Bayes model.  
  
# Define a vectoriser  
vectorizer = CountVectorizer()  
X_train_counts = vectorizer.fit_transform(X_train)  
  
# Train the Multinomial Naive Bayes classifier  
clf = MultinomialNB()  
clf.fit(X_train_counts, y_train)  
  
[69] ✓ 0.2s  
...  
MultinomialNB()  
MultinomialNB()
```

The data for the LSTM model is pre-processed in preparation for input. The independent variables are tokenised and converted into sequences while the dependent variable is passed through a one-hot label encoder.

The output shape of the model is determined by the unique labels within the dependent variables.



The screenshot shows a Jupyter Notebook interface with a dark theme. The left sidebar displays the Explorer and Open Editors panels. The main editor area is titled '4. Preprocess data.' and contains the following Python code:

```
# Define Tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)

# Define max_len
max_len = 100 # Adjust as needed

# Tokenize and pad sequences
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)
vocab_size = len(tokenizer.word_index) + 1
X_train_pad = pad_sequences(X_train_seq, maxlen=max_len, padding='post')
X_test_pad = pad_sequences(X_test_seq, maxlen=max_len, padding='post')

# Combine training and test sets for Label encoding
combined_labels = pd.concat([y_train, y_test])

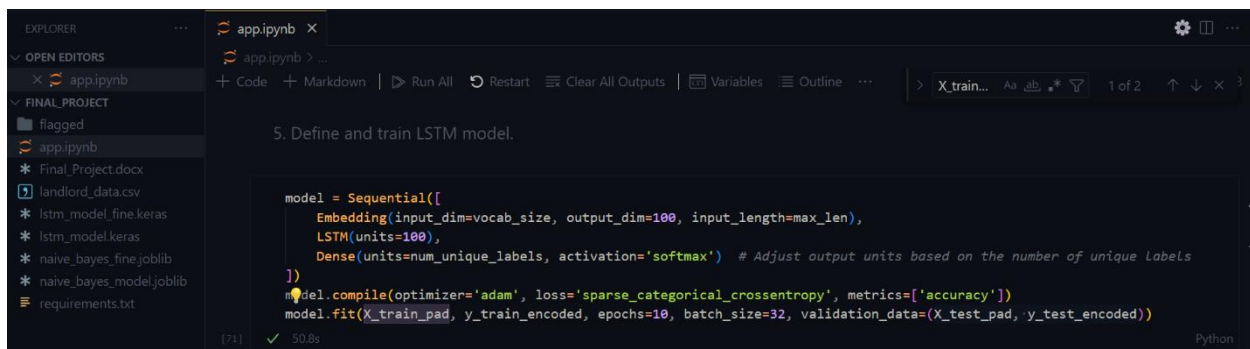
# Initialize LabelEncoder
label_encoder = LabelEncoder()
label_encoder.fit(combined_labels)

# Fit Label encoder and transform labels
y_train_encoded = label_encoder.transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

num_unique_labels = len(set(combined_labels))
```

The code is executed, and the output shows the number of unique labels as 10. The status bar at the bottom indicates the execution time as 0.1s.

We define the LSTM model and the corresponding neural network layers from the Keras library in TensorFlow.



The screenshot shows a Jupyter Notebook interface with a dark theme. The left sidebar displays the Explorer and Open Editors panels. The main editor area is titled '5. Define and train LSTM model.' and contains the following Python code:

```
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=100, input_length=max_len),
    LSTM(units=100),
    Dense(units=num_unique_labels, activation='softmax') # Adjust output units based on the number of unique labels
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train_pad, y_train_encoded, epochs=10, batch_size=32, validation_data=(X_test_pad, y_test_encoded))
```

The code is executed, and the output shows the model's performance metrics. The status bar at the bottom indicates the execution time as 50.8s.

Save local instances of both models.

```
6. Save local instances of the models

# Save the Naive Bayes classifier
joblib.dump(clf, "naive_bayes_model.joblib")

# Load the Naive Bayes classifier
clf = joblib.load("naive_bayes_model.joblib")

# Save LSTM Model
model.save("lstm_model.keras")
```

[72] ✓ 0.2s Python

Evaluate the performance of both models.

```
7. Evaluate Both model's performance

# Evaluate Naive Bayes model
X_val_counts = vectorizer.transform(X_val)
y_val_pred_nb = clf.predict(X_val_counts)
accuracy = accuracy_score(y_val, y_val_pred_nb)
print("Test Accuracy:", accuracy)

... Test Accuracy: 0.012578616352201259

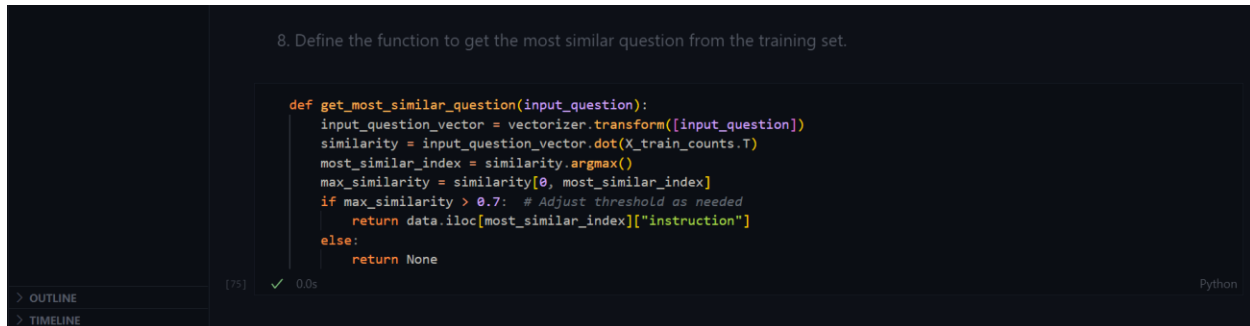
# Evaluate LSTM model
X_val_pad = pad_sequences(tokenizer.texts_to_sequences(X_val), maxlen=max_len, padding='post')
y_val_pred_lstm = model.predict(X_val_pad)
y_val_pred_lstm_classes = label_encoder.inverse_transform(y_val_pred_lstm.argmax(axis=-1))
loss, accuracy = model.evaluate(X_test_pad, y_test_encoded)
print("LSTM Model Performance:")
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)

... 15/15 1s 44ms/step
19/19 1s 38ms/step - accuracy: 0.0042 - loss: 11.2447
LSTM Model Performance:
Test Loss: 11.22800350189209
Test Accuracy: 0.005025125574320555
```

[74] ✓ 1.8s Python

## 4) Define Functions.

Define the function to get the most similar question from the training set. This function works by vectorising the user's question and looks for questions with at least a 70% match in vector shape as the preprocessed data.

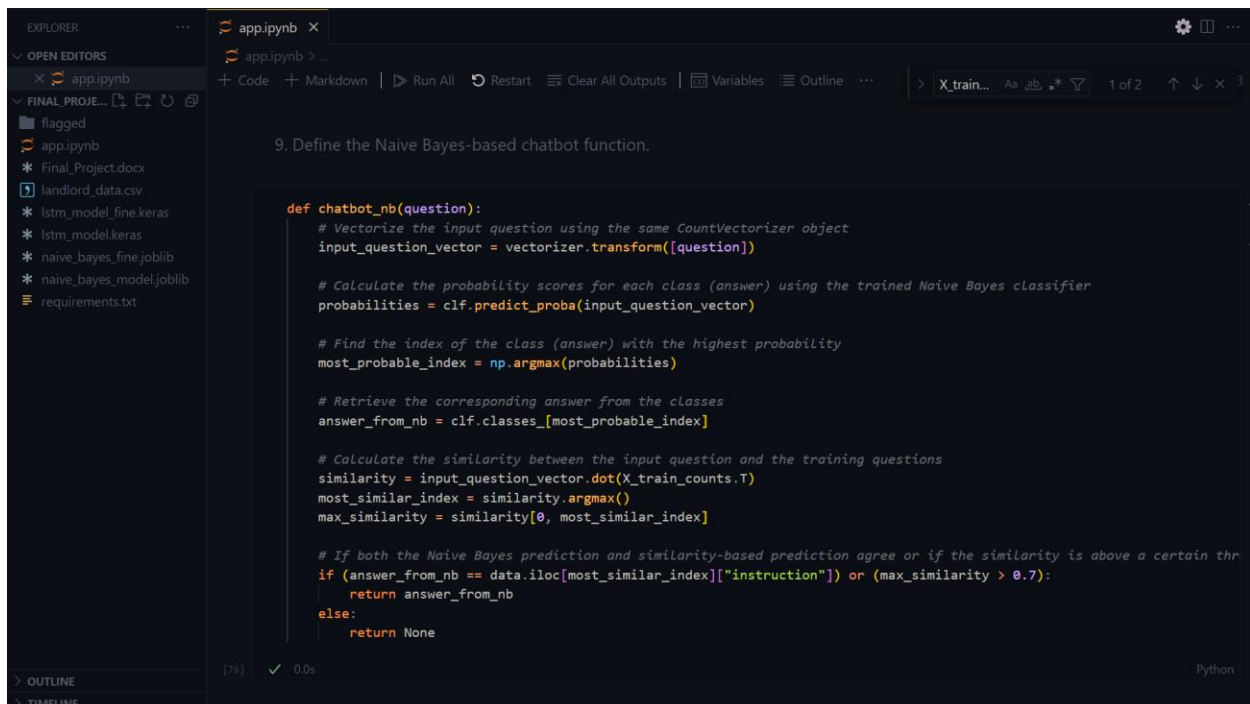


8. Define the function to get the most similar question from the training set.

```
def get_most_similar_question(input_question):  
    input_question_vector = vectorizer.transform([input_question])  
    similarity = input_question_vector.dot(X_train_counts.T)  
    most_similar_index = similarity.argmax()  
    max_similarity = similarity[0, most_similar_index]  
    if max_similarity > 0.7: # Adjust threshold as needed  
        return data.iloc[most_similar_index]["instruction"]  
    else:  
        return None
```

[75] ✓ 0.0s Python

Define the Naïve Bayes chatbot.



9. Define the Naive Bayes-based chatbot function.

```
def chatbot_nb(question):  
    # Vectorize the input question using the same CountVectorizer object  
    input_question_vector = vectorizer.transform([question])  
  
    # Calculate the probability scores for each class (answer) using the trained Naive Bayes classifier  
    probabilities = clf.predict_proba(input_question_vector)  
  
    # Find the index of the class (answer) with the highest probability  
    most_probable_index = np.argmax(probabilities)  
  
    # Retrieve the corresponding answer from the classes  
    answer_from_nb = clf.classes_[most_probable_index]  
  
    # Calculate the similarity between the input question and the training questions  
    similarity = input_question_vector.dot(X_train_counts.T)  
    most_similar_index = similarity.argmax()  
    max_similarity = similarity[0, most_similar_index]  
  
    # If both the Naive Bayes prediction and similarity-based prediction agree or if the similarity is above a certain threshold  
    if (answer_from_nb == data.iloc[most_similar_index]["instruction"]) or (max_similarity > 0.7):  
        return answer_from_nb  
    else:  
        return None
```

[76] ✓ 0.0s Python

Define the LSTM chatbot.

```
10. Define the LSTM-based chatbot function.

def chatbot_lstm(question):
    input_sequence = tokenizer.texts_to_sequences([question])
    input_sequence_pad = pad_sequences(input_sequence, maxlen=max_len, padding='post')
    prediction = model.predict(input_sequence_pad)
    predicted_class = label_encoder.inverse_transform([prediction.argmax()])[0]
    return predicted_class
```

## 5) Fine-tuning.

Define a function to fine-tune the Naïve Bayes Model. This is done by retraining the model on the dataset.

```
# Fine-tune Naive Bayes model
def fine_tune_naive_bayes(X_train_new, y_train_new):
    # Load the saved Naive Bayes classifier
    clf = joblib.load("naive_bayes_model.joblib")

    # Update the model with new data
    # X_train_new_counts = vectorizer.transform(X_train_new)
    clf.partial_fit(X_train_new, y_train_new, classes=np.unique(y_train_new))

    # Save the fine-tuned model
    joblib.dump(clf, "naive_bayes_fine.joblib")

# Fine-tune the Naive Bayes model with additional data
fine_tune_naive_bayes(X_train_counts, y_train)
```



Define a function to fine-tune the Long Short-Term Memory (LSTM) model. This is done by retraining the model on the dataset.

```
def fine_tune_lstm(X_train_new, y_train_new):
    # Load the saved LSTM model
    model = tf.keras.models.load_model("lstm_model.keras")

    # Preprocess the new data
    X_train_new_seq = tokenizer.texts_to_sequences(X_train_new)
    X_train_new_pad = pad_sequences(X_train_new_seq, maxlen=max_len, padding='post')

    # Fine-tune the model
    model.fit(X_train_new, y_train_new, epochs=5, batch_size=32, validation_split=0.1)

    # Save the fine-tuned model
    model.save("lstm_model_fine.keras")

# Fine-tune the LSTM model with additional data
fine_tune_lstm(X_test_pad, y_test_encoded)
```

Epoch 1/5  
17/17 ————— 4s 109ms/step - accuracy: 0.0000e+00 - loss: 9.7133 - val\_accuracy: 0.0000e+00 - val\_loss: 7.2741  
Epoch 2/5  
17/17 ————— 2s 110ms/step - accuracy: 0.0000e+00 - loss: 6.8575 - val\_accuracy: 0.0000e+00 - val\_loss: 9.0272  
Epoch 3/5  
17/17 ————— 2s 105ms/step - accuracy: 9.9122e-04 - loss: 6.5712 - val\_accuracy: 0.0000e+00 - val\_loss: 9.5163  
Epoch 4/5  
17/17 ————— 2s 90ms/step - accuracy: 0.0000e+00 - loss: 6.5752 - val\_accuracy: 0.0000e+00 - val\_loss: 9.7205  
Epoch 5/5  
17/17 ————— 1s 68ms/step - accuracy: 0.0054 - loss: 6.4956 - val\_accuracy: 0.0000e+00 - val\_loss: 9.6865

Evaluate the accuracy of the fine-tuned models.

```
# Evaluate Naive Bayes model
X_val_counts = vectorizer.transform(X_val)
y_val_pred_nb = clf.predict(X_val_counts)
accuracy = accuracy_score(y_val, y_val_pred_nb)
print("Test Accuracy:", accuracy)

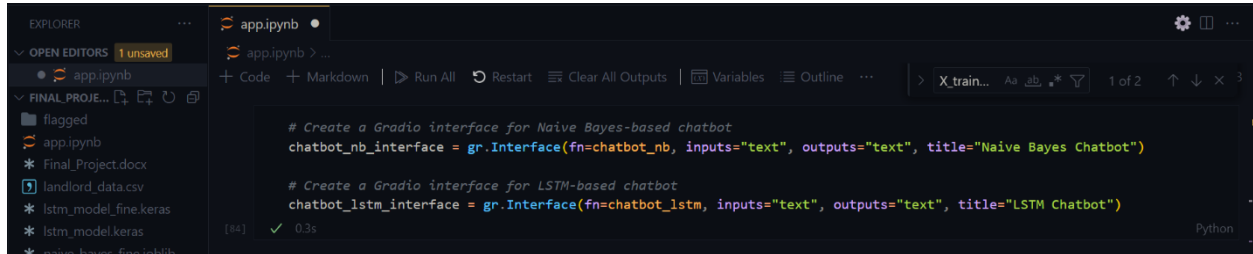
# Evaluate LSTM model
X_val_pad = pad_sequences(tokenizer.texts_to_sequences(X_val), maxlen=max_len, padding='post')
y_val_pred_lstm = model.predict(X_val_pad)
y_val_pred_lstm_classes = label_encoder.inverse_transform(y_val_pred_lstm.argmax(axis=-1))
loss, accuracy = model.evaluate(X_test_pad, y_test_encoded)
print("LSTM Model Performance:")
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

Test Accuracy: 0.012578616352201259

LSTM Model Performance:  
Test Loss: 11.22800350189209  
Test Accuracy: 0.005025125574320555

## 6) User Interfaces.

Define both model chatbots using the Gradio library in Python.



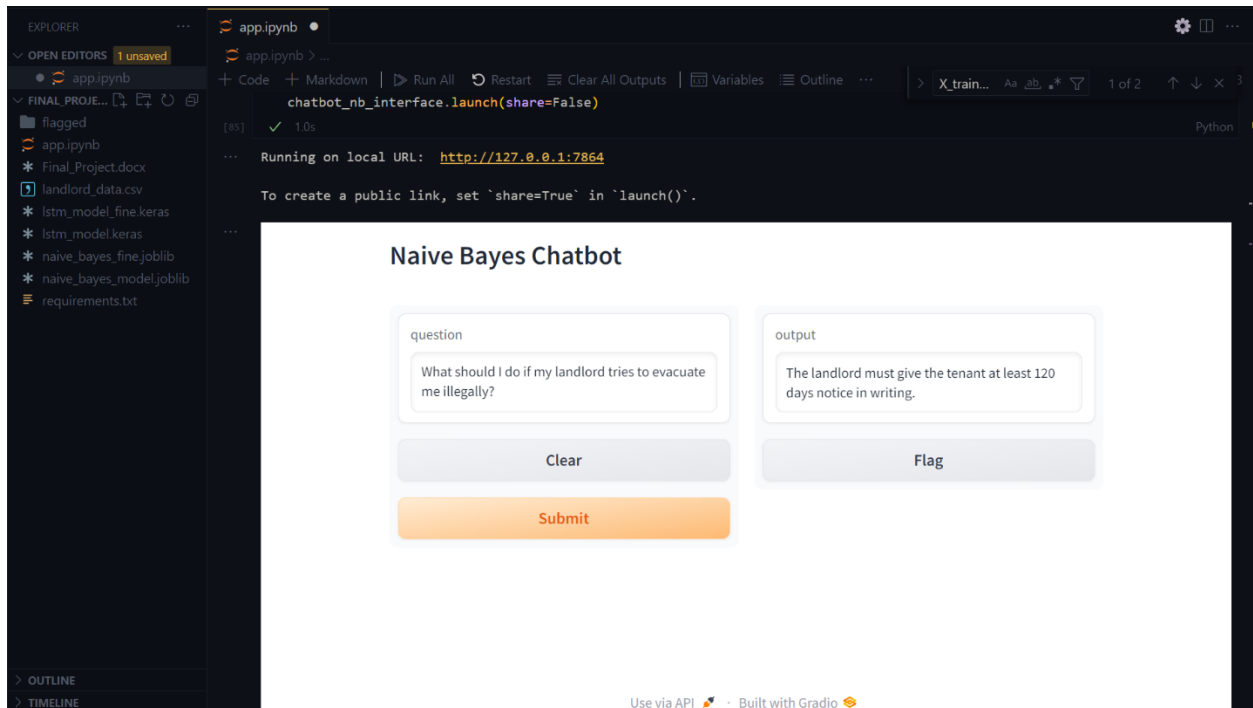
```
app.ipynb
app.ipynb > ...
+ Code + Markdown | ▶ Run All ⏮ Restart ⏹ Clear All Outputs | 📄 Variables 📄 Outline ...
> X_train... Aa _b_ * 1 of 2 ↑ ↓ × 3

# Create a Gradio interface for Naive Bayes-based chatbot
chatbot_nb_interface = gr.Interface(fn=chatbot_nb, inputs="text", outputs="text", title="Naive Bayes Chatbot")

# Create a Gradio interface for LSTM-based chatbot
chatbot_lstm_interface = gr.Interface(fn=chatbot_lstm, inputs="text", outputs="text", title="LSTM Chatbot")

[84] ✓ 0.3s Python
```

Load the NB Chatbot. The chatbot can be accessed either through the Jupyter Notebook or through localhost using the generated URL.



```
app.ipynb
app.ipynb > ...
+ Code + Markdown | ▶ Run All ⏮ Restart ⏹ Clear All Outputs | 📄 Variables 📄 Outline ...
> X_train... Aa _b_ * 1 of 2 ↑ ↓ × 3

chatbot_nb_interface.launch(share=False)

[85] ✓ 1.0s Python

... Running on local URL: http://127.0.0.1:7864

To create a public link, set 'share=True' in 'launch()'.

...

Naive Bayes Chatbot

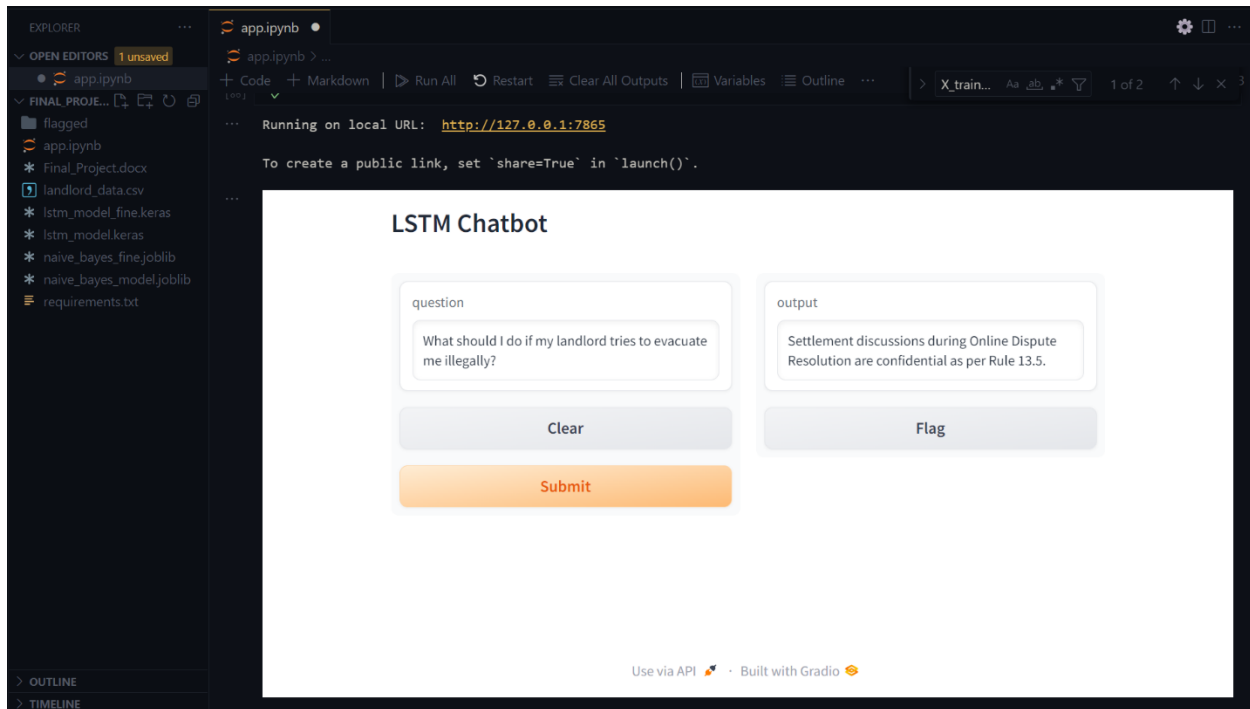
question
What should I do if my landlord tries to evacuate me illegally?

output
The landlord must give the tenant at least 120 days notice in writing.

Clear Flag Submit

Use via API - Built with Gradio
```

Load the LSTM chatbot. The chatbot can be accessed either through the Jupyter Notebook or through localhost using the generated URL.



## **OUR SUGGESTION:**

Over our interactions with the ML models via the Chatbots, we found that the Multinomial Naïve Bayes Chatbot performed better than the Long Short-Term Memory (LSTM) chatbot.

Both models could generate responses, but the NB Chatbot is the model that returns information much more closely related to the dataset the models were trained on.