BENHA NATIONAL UNIVERSITY

COMPUTER SCIENCE

Course Title: **Deep Learning**

Shakespeare Text Generation using VAE + Transformer

SUBMITTED BY

Student Name	Student ID	Section	Grade
أحمد سعد كرم السيد	*22030012*	*1*	
مروان اشرف حسن مجاهد	*22030173*	*2*	
أمل أشرف علي أبوالعنين	*22030036*	*1*	
رغد هاني طلعت احمد	*22030059*	*1*	

Submitted to: *[Eng: Hossam Fares /Mohamed Bahgat]*

Dr. Amr Nagy

Date: December 23, 2025

Shakespeare Text Generation VAE+Transformer

Team Names	Code	Section
Raghad Hany Talaat	22030059	Section 1
Ahmed Saad Karam	22030012	Section 1
Aml Ashraf Ali	22030036	Section 1
Marawan Ashraf Hassan	22030173	Section 2

▼ Introduction

Project Idea

This project implements a **deep generative language model** capable of producing **Shakespeare-style text** by combining two powerful paradigms:

- **Variational Autoencoders (VAE)** → to learn a smooth, structured *latent semantic space* of sentences.
- **Transformer-based decoding** → to generate fluent, context-aware text sequences.

Unlike traditional language models that generate text token-by-token without an explicit semantic representation, this model **encodes entire sentences into a continuous latent vector** and then decodes them back into text. This enables:

- Controlled generation
- Sentence interpolation
- Latent space visualization
- Style-consistent text synthesis

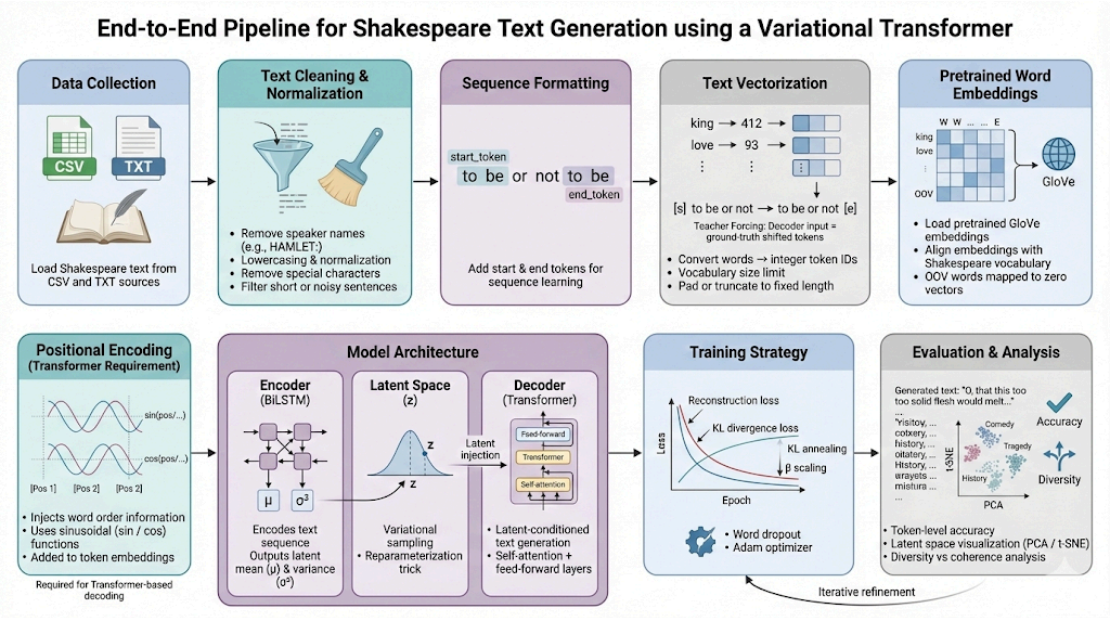
Project Goals

The main objectives of this project are:

- Learn **semantic representations** of Shakespearean sentences
- Generate **coherent and stylistically consistent text**
- Enable **latent space interpolation** between sentences
- Combine **VAE regularization** with **Transformer expressiveness**
- Evaluate diversity, coherence, and semantic structure of generated text

---

Overall Pipeline Overview



▼ 🧹 Data Pre-Processing & Embedding Layer

Environment Setup & Imports

```
import tensorflowas tf
import numpyas np
import pandasas pd
import re
import os
import zipfile
from tensorflow.kerasimport layers
from tqdm.notebookimport tqdm
```

Purpose of Each Import

Library	Role
TensorFlow	Deep learning framework used to build and train the VAE-Transformer model
NumPy	Numerical operations, especially for embedding matrices
Pandas	Loading and processing CSV-based Shakespeare datasets
re	Regular expressions for text cleaning
os	File existence checks and path handling
zipfile	Extracting downloaded GloVe embeddings
Keras Layers	Neural network building blocks
tqdm	Training progress visualization

Downloading GloVe Word Embeddings

```
ifnot os.path.exists('glove.6B.100d.txt'):
    !wget http://nlp.stanford.edu/data/glove.6B.zip
    !unzip -q glove.6B.zip
```

- Checks if the **100-dimensional GloVe embeddings** already exist locally
- Downloads vectors trained on **6 billion tokens**
- Each word is represented as a **100-dimensional semantic vector**

Why GloVe?

- Captures **semantic relationships** between words

- Provides strong linguistic priors
- Speeds up convergence compared to random embeddings

Data Loading

Defining Data Sources

```
csv_path ="/content/Shakespeare_data.csv"
txt_path ="/content/alllines.txt"
```

The project supports **two data formats**:

- **CSV** → structured dialogue dataset
- **TXT** → raw Shakespeare text

Loading the Data

```
texts = []
```

Stores all raw textual samples.

```
df = pd.read_csv(csv_path)
texts.extend(df["PlayerLine"].dropna().tolist())
```

- Loads spoken dialogue only
- Removes missing entries

```
withopen(txt_path,"r", encoding="utf-8", errors="ignore")as f:
    texts.extend(f.readlines())
```

- Reads raw text line-by-line
- Ignores corrupted characters

Text Cleaning Function

```
def clean_text(t):
```

Defines a standardized preprocessing pipeline.

```
t = t.lower()
```

- Reduces vocabulary size
- Improves embedding coverage

```
if":"in tandlen(t.split(":",1)[0]) <20:
    t = t.split(":",1)[1]
```

- Removes speaker names such as:

  ```
  HAMLET:To beornotto be
  ```

```
t = re.sub(r"[^a-z0-9\s,.!?]","", t)
```

Removes:

- Stage directions
- Archaic symbols
- Special characters

```
t = re.sub(r"\s+"," ", t)
return t.strip()
```

- Normalizes whitespace
- Removes leading/trailing spaces

Sentence Filtering & Deduplication

```
cleaned_texts = [clean_text(t)for tin textsiflen(t.split()) >5]
```

- Discards short or meaningless sentences
- Ensures semantic richness

```
unique_texts =list(set(cleaned_texts))
```

- Removes duplicate lines
- Prevents memorization during training

Adding Start & End Tokens

```
formatted_texts = [f"start_token {t} end_token"for tin unique_texts]
```

Why This Is Critical

- Enables **teacher forcing**
- Allows model to learn:
  - Sentence boundaries
  - When to stop generation
- Required for autoregressive decoding

Text Vectorization

Hyperparameters

```
VOCAB_SIZE =20000
SEQ_LEN =35
```

- Vocabulary limited to most frequent 20k tokens
- Sentences padded/truncated to 35 words

Vectorization Layer

```
vectorizer = layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_mode="int",
    output_sequence_length=SEQ_LEN +1,
    standardize=None
)
```

Parameter	Purpose
max_tokens	Vocabulary cap
output_mode	Word → integer
sequence_length	Extra token for shifting
standardize=None	Custom cleaning already applied

```
vectorizer.adapt(formatted_texts)
```

- Learns vocabulary distribution from Shakespeare text

```
vocab = vectorizer.get_vocabulary()
```

- Stores token ↔ word mapping
- Required for decoding generated text

Dataset Construction (Teacher Forcing)

```
defprepare_batch(text):
    seq = vectorizer(text)
```

Converts sentence into token IDs.

```
enc_in = seq[:-1]
dec_in = seq[:-1]
target = seq[1:]
```

Component	Role
Encoder Input	Full sentence representation
Decoder Input	Shifted right sequence
Target	Shifted left prediction

This setup allows:

- Parallel decoding
- Stable training
- Faster convergence

```
dataset = tf.data.Dataset.from_tensor_slices(formatted_texts)
dataset = dataset.map(prepare_batch)
dataset = dataset.shuffle(4096).batch(64).prefetch()
```

Optimized pipeline using:

- Shuffling for generalization
- Batching for GPU efficiency
- Prefetching for performance

Loading GloVe Embeddings

```
defload_glove_embeddings(vocab, embed_dim=100):
```

Creates an embedding matrix aligned with the model vocabulary.

```
embeddings_index = {}
```

Stores pretrained vectors as:

```
word → embedding
```

```
withopen("glove.6B.100d.txt"):
```

Reads each GloVe vector line-by-line.

```
embedding_matrix = np.zeros((num_tokens, embed_dim))
```

- Shape: `(vocab_size, embedding_dim)`
- Words not found → zero vectors

```
embedding_matrix[i] = embedding_vector
```

- Aligns **TextVectorization indices** with GloVe vectors
- Ensures correct word-embedding mapping

```
print(f"Converted {hits} words ({misses} misses)")
```

Reports vocabulary coverage.

▼ 🧠 Model Architecture

This section describes the **core neural architecture and optimization strategy** used in the project.

The model is a **Text Variational Autoencoder (TextVAE)** that combines:

- **Pretrained word embeddings (GloVe)**
- **Bidirectional LSTM encoder**
- **Transformer-based decoder**
- **Variational latent space modeling**

The goal is to generate **coherent, diverse Shakespeare-like sentences** by learning a **continuous latent representation** of text while preserving linguistic structure and semantics.

🧭 Architecture Overview

Input Text → Encoder → Latent Space → Decoder → Generated Text

- The **encoder** compresses a sentence into a probabilistic latent vector.
- The **decoder** reconstructs text from this latent representation using attention mechanisms.
- **Variational inference** ensures smoothness and diversity in generated outputs.

Positional Encoding

❓ Why Positional Encoding Is Needed

Transformers do **not** inherently understand word order because they rely entirely on **self-attention**, not recurrence or convolution.

To inject **sequence order information**, positional encodings are added to word embeddings.

🔧 Function Definition

```
defpositional_encoding(seq_len, d_model):
```

- `seq_len`: Maximum sequence length
- `d_model`: Embedding dimensionality

🔢 Position & Dimension Indices

```
pos = np.arange(seq_len)[:,None]
```

- Creates a column vector of word positions

  (0, 1, 2, ..., seq_len − 1)

```
i = np.arange(d_model)[None, :].astype(np.float32)
```

- Creates a row vector of embedding dimensions

  (0, 1, 2, ..., d_model − 1)

📐 Angle Rate Computation

```
angle_rates =1 / np.power(10000, (2 * (i //2)) / d_model)
```

- Computes different frequencies for each embedding dimension
- Even and odd indices share the same frequency
- Enables the model to represent **relative position information**

📊 Angle Matrix

```
angle_rads = pos * angle_rates
```

- Combines word position with frequency scaling
- Produces a `(seq_len × d_model)` matrix

🌊 Sinusoidal Encoding

```
angle_rads[:,0::2] = np.sin(angle_rads[:,0::2])
angle_rads[:,1::2] = np.cos(angle_rads[:,1::2])
```

- **Even indices → sine**
- **Odd indices → cosine**
- Guarantees:
  - Unique encoding for each position
  - Generalization to longer sequences

🔁 Final Tensor Formatting

```
return tf.cast(angle_rads[None, ...], tf.float32)
```

- Adds batch dimension
- Converts to TensorFlow tensor

✅ Result

A **fixed, non-trainable positional signal** that is added to token embeddings inside the decoder.

TextVAE Model Architecture

📦 Class Definition

```
classTextVAE(tf.keras.Model):
```

Defines a **custom Keras model** implementing a **Variational Autoencoder for text**.

🧩 Model Initialization

```
def__init__(self, vocab_size, seq_len, latent_dim, embed_dim, embedding_matrix):
```

Parameters

- `vocab_size`: Size of the vocabulary
- `seq_len`: Maximum sequence length
- `latent_dim`: Dimensionality of latent space
- `embed_dim`: Word embedding size
- `embedding_matrix`: Pretrained GloVe weights

🔤 Embedding Layer (Pretrained & Trainable)

```
self.embedding = layers.Embedding(
    vocab_size,
    embed_dim,
    embeddings_initializer=Constant(embedding_matrix),
    trainable=True
)
```

Explanation

- Initializes embeddings using **GloVe**
- `trainable=True` allows fine-tuning
- Adapts generic embeddings to **Shakespearean language style**

🧠 Encoder (Variational)

Bidirectional LSTM

```
self.encoder_lstm = layers.Bidirectional(layers.LSTM(256))
```

- Reads text **forward and backward**
- Captures long-range dependencies
- Produces a fixed-length sentence representation

Latent Parameter Projection

```
self.encoder_dense = layers.Dense(latent_dim *2)
```

- Outputs a vector twice the latent size
- Later split into:
  - **Mean (μ)**
  - **Log-variance (log σ^2)**

🔄 Encoding Step

```
def encode(self, x):
```

```
x =self.embedding(x)
```

- Converts token IDs → embeddings

```
x =self.encoder_lstm(x)
```

- Encodes entire sequence

```
mean, logvar = tf.split(self.encoder_dense(x),2, axis=1)
```

- Separates distribution parameters

🎲 Reparameterization Trick

```
def reparameterize(self, mean, logvar):
```

```
eps = tf.random.normal(shape=tf.shape(mean))
```

- Samples random noise

```
return eps * tf.exp(0.5 * logvar) + mean
```

Purpose

- Enables **backpropagation through stochastic sampling**
- Core mechanism of **Variational Autoencoders**

🔁 Decoder Architecture (Transformer-Based)

Latent Projection

```
self.latent_projection = layers.Dense(seq_len * embed_dim)
```

- Expands latent vector
- Aligns it with sequence length

Positional Encoding

```
self.pos_encoding = positional_encoding(seq_len, embed_dim)
```

- Adds word order information

Transformer Blocks

```
self.decoder_layers = [
    layers.MultiHeadAttention(num_heads=4, key_dim=embed_dim)
]
```

- Enables **context-aware token generation**

Feed-Forward Networks

```
self.ffn_layers = [
    Dense(512, activation="relu"),
    Dense(embed_dim)
]
```

- Applies non-linear transformations

- Increases model expressiveness

Layer Normalization

```
self.layernorm1,self.layernorm2
```

- Stabilizes training
- Improves convergence

🚫 Causal Masking

```
mask = tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1,0)
```

- Prevents attention to **future tokens**
- Ensures autoregressive generation

🎯 Output Layer

```
self.output_layer = layers.Dense(vocab_size)
```

- Converts hidden states → token logits

⚙️ Optimization & Learning Rate Strategy

📏 Steps per Epoch (Safe Handling)

```
steps_per_epoch = len(dataset)
```

- Dynamically computed
- Fallback mechanism prevents runtime errors

📉 Cosine Learning Rate Decay

```
lr_schedule = CosineDecay(
    initial_learning_rate=0.001,
    decay_steps=30 * steps_per_epoch,
```

```
    alpha=0.1
)
```

Benefits

- Smooth learning rate reduction
- Prevents early convergence
- Improves generalization

🔧 Optimizer Configuration

```
optimizer = Adam(
    learning_rate=lr_schedule,
    clipnorm=1.0
)
```

- **Adam optimizer** for stability
- **Gradient clipping** prevents exploding gradients

✅ Summary of the architecture

This architecture combines:

- **Statistical latent modeling (VAE)**
- **Sequential understanding (BiLSTM)**
- **Parallel attention-based decoding (Transformer)**

Resulting in a model capable of:

- Generating **diverse**
- **Coherent**
- **Stylistically faithful** Shakespeare-like text

▼ 🏋️ Model Training

The training process is carefully designed to balance:

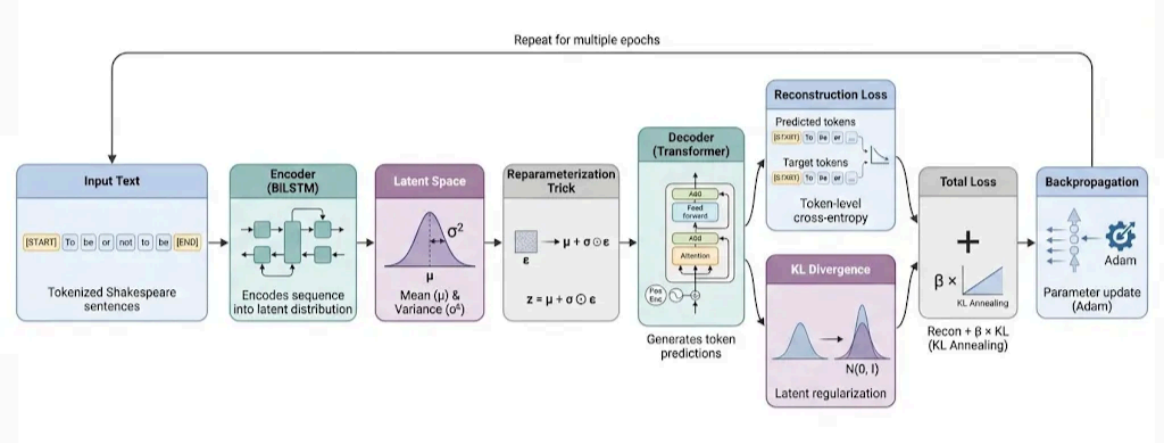
- **Text reconstruction quality**
- **Latent space regularization**
- **Training stability**

To achieve this, the model integrates:

- **Cosine learning rate decay**
- **KL annealing (warm-up)**
- **KL loss clipping**
- **Word dropout**
- **Gradient clipping**

Together, these techniques ensure stable convergence and prevent common failure modes in text VAEs, such as **posterior collapse**.

🧠Training Process Overview



This process allows the model to:

- Learn meaningful latent representations
- Generate diverse yet coherent text
- Avoid ignoring the latent space

🔧 Learning Rate Scheduler

Code

```
lr_schedule = tf.keras.optimizers.schedules.CosineDecay(
    initial_learning_rate=0.001,
    decay_steps=30 *len(dataset),
    alpha=0.1
)
```

- **CosineDecay** gradually reduces the learning rate following a cosine curve

- `initial_learning_rate=0.001`

  → High starting rate encourages faster early learning
- `decay_steps=30 * len(dataset)`

  → Learning rate decays smoothly over all training steps (30 epochs)
- `alpha=0.1`

  → Final learning rate is 10% of the initial rate

Why Cosine Decay?

- Prevents abrupt learning rate drops
- Encourages smooth convergence
- Improves final generalization performance

🔧 Optimizer Configuration

```
optimizer = tf.keras.optimizers.Adam(
    learning_rate=lr_schedule,
    clipnorm=1.0
)
```

- **Adam optimizer** adapts learning rates per parameter
- `clipnorm=1.0` prevents exploding gradients
- Essential for stability when training:
  - Transformers
  - VAEs
  - Long sequences

🔁 Training Loop Configuration

```
EPOCHS =30
KL_WARMUP =5
```

Meaning

- **30 epochs** allow sufficient convergence
- **KL warm-up = 5 epochs**

- KL loss influence starts small
- Gradually increases to avoid posterior collapse

⚙️ Single Training Step

Function Definition

```
@tf.function
def train_step(inputs, target, beta, word_dropout):
```

Purpose

- Compiles the function into a TensorFlow graph
- Improves execution speed and performance

🔓 Input Unpacking

```
enc_in, dec_in = inputs
```

- `enc_in`: Encoder input sequence
- `dec_in`: Decoder input (shifted right)
- `target`: Expected output tokens

Gradient Tape

```
with tf.GradientTape()as tape:
```

- Tracks all operations for automatic differentiation
- Required to compute gradients

🧠 Encoding Step

```
mean, logvar = model.encode(enc_in)
```

- Encoder outputs:
  - **Mean (μ)**

- **Log-variance (log σ^2)**

These define the latent Gaussian distribution.

🎲 Latent Sampling (Reparameterization)

```
z = model.reparameterize(mean, logvar)
```

- Samples latent vector while preserving gradient flow
- Core VAE mechanism

🔄 Decoding Step

```
logits = model.decode(
    z, dec_in,
    training=True,
    word_dropout_rate=word_dropout
)
```

- Generates token logits
- Applies **word dropout**:
  - Randomly masks decoder inputs
  - Forces reliance on latent variable `z`
  - Improves robustness

📉 Reconstruction Loss

```
recon_loss = tf.reduce_mean(
    tf.keras.losses.sparse_categorical_crossentropy(
        target, logits, from_logits=True
    )
)
```

- Measures how well predicted tokens match ground truth
- Computed at the **token level**
- Encourages grammatical and semantic correctness

KL Divergence Loss

```
kl_loss = -0.5 * tf.reduce_mean(
    tf.reduce_sum(
        1 + logvar - tf.square(mean) - tf.exp(logvar),
        axis=1
    )
)
```

Purpose

- Regularizes latent space
- Encourages latent distribution to match standard normal
- Enables smooth interpolation & sampling

KL Loss Clipping (Hinge Loss)

```
kl_loss_clipped = tf.maximum(kl_loss,0.1)
```

- Prevents KL loss from collapsing to zero
- Forces the model to **use latent variables**
- Mitigates posterior collapse

➕ Total Loss

```
total_loss = recon_loss + (beta * kl_loss_clipped)
```

Interpretation

- `beta` controls KL importance
- Balances:
  - Reconstruction accuracy
  - Latent regularization

🔁 Backpropagation

```
grads = tape.gradient(total_loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

- Computes gradients
- Updates model parameters
- Uses gradient clipping for stability

KL Annealing Strategy

```
if epoch < KL_WARMUP:
    BETA = (epoch / KL_WARMUP) *0.5
else:
    BETA =0.5
```

- KL weight increases gradually over first 5 epochs
- Prevents early dominance of KL loss
- Encourages encoder to learn meaningful representations first

---

📊 Epoch-Level Tracking

```
epoch_loss += loss
epoch_recon += recon
epoch_kl += kl
```

- Accumulates metrics across batches
- Enables detailed training diagnostics

📈 Progress Monitoring

```
progress = tqdm(dataset, desc=f"Epoch {epoch+1}")
```

- Visual progress bar
- Displays:
  - Total loss
  - Reconstruction loss
  - KL loss

📌 Epoch Summary

```
print(f"Epoch {epoch+1} | Loss: ... | Recon: ... | KL: ...")
```

- Reports average metrics per epoch
- Used to:
  - Monitor convergence
  - Detect instability
  - Compare experiments

```
Epoch 28 | Loss: 1.2423 | Recon: 1.1922 | KL: 0.0891

Epoch 29 | Loss: 1.2389 | Recon: 1.1888 | KL: 0.0894

Epoch 30 | Loss: 1.2363 | Recon: 1.1862 | KL: 0.0889
Training Complete!
```

✅ Final Outcome

This training strategy ensures that the model:

- Learns **meaningful latent representations**
- Generates **diverse yet coherent Shakespeare-style text**
- Avoids common VAE failures
- Trains stably with large vocabularies and long sequences

▼ 📊Evaluation, Visualization, Generation, and Analysis

This section presents the **post-training evaluation and analysis pipeline** of the proposed **TextVAE with Transformer Decoder**.

It focuses on quantitatively measuring model performance, qualitatively inspecting latent representations, and evaluating generative behavior in terms of **diversity, coherence, and controllability**.

▼ 1️⃣ Model Evaluation (Token-Level Metrics)

Objective

To quantitatively evaluate how accurately the model predicts tokens during decoding, independent of sentence-level fluency.

Imported Metrics

```
from sklearn.metricsimport accuracy_score, precision_score, recall_score
```

- **Accuracy**: Overall proportion of correctly predicted tokens
- **Precision (Macro)**: Measures correctness across all vocabulary tokens equally
- **Recall (Macro)**: Measures coverage of ground-truth tokens equally

Macro averaging is used to avoid dominance of frequent tokens and better reflect **lexical diversity**.

These metrics are computed **at the token level**, which is appropriate for sequence generation tasks.

Evaluation Function

```
def evaluate_model(model, dataset, max_batches=50):
```

Evaluates the model on a subset of the dataset to ensure:

- Fast evaluation
- Representative performance estimation

Step-by-Step Process

1. Initialize storage

```
y_true = []
y_pred = []
```

Stores ground-truth tokens and predicted tokens.

---

2. Iterate over dataset

```
for i, batchinenumerate(dataset):
if i >= max_batches:
break
```

Limits evaluation to a fixed number of batches for efficiency.

3. Unpack dataset batch

```
(enc_in, dec_in), target = batch
```

Dataset structure:

- `enc_in`: encoder input
- `dec_in`: decoder input (teacher forcing)
- `target`: expected output tokens

4. Forward pass

```
mean, logvar = model.encode(enc_in)
z = model.reparameterize(mean, logvar)
logits = model.decode(z, dec_in, training=False)
```

- Disables word dropout
- Ensures deterministic evaluation

5. Token prediction

```
predictions = tf.argmax(logits, axis=-1)
```

Selects the most probable token at each timestep.

6. Flatten tokens

```
y_true.extend(target.numpy().flatten())
y_pred.extend(predictions.numpy().flatten())
```

Required because `sklearn` metrics expect 1D arrays.

7. Remove padding tokens

```
mask = y_true !=0
```

Padding tokens (`PAD = 0`) are excluded to avoid metric distortion.

8. Compute metrics

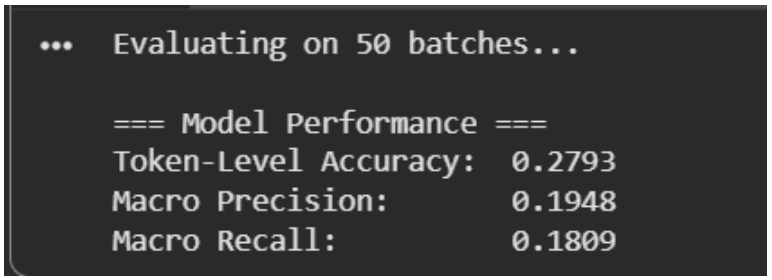
```
accuracy_score
precision_score (macro)
```

```
recall_score (macro)
```

- **Accuracy** → overall correctness
- **Macro Precision / Recall** → treats all tokens equally (important for vocabulary diversity)

📌 Evaluation Results

> Token-level Accuracy, Precision, and Recall Results



▼ 2️⃣ Latent Space Visualization (PCA)

Objective

To inspect whether the model learns a **structured and semantically meaningful latent space**.

2.1 PCA Visualization

Encoding a subset of sentences

```
subset_texts = unique_texts[:500]
subset_vecs = vectorizer(np.array(subset_texts))
```

- Selects 500 sentences for visualization
- Converts text into token sequences

Encoding sentences

```
mean, logvar = model.encode(subset_vecs)
z_subset = model.reparameterize(mean, logvar)
```

Maps sentences into the latent space.

Dimensionality reduction

```
PCA(n_components=2)
```

Reduces 128D latent vectors to 2D for visualization.

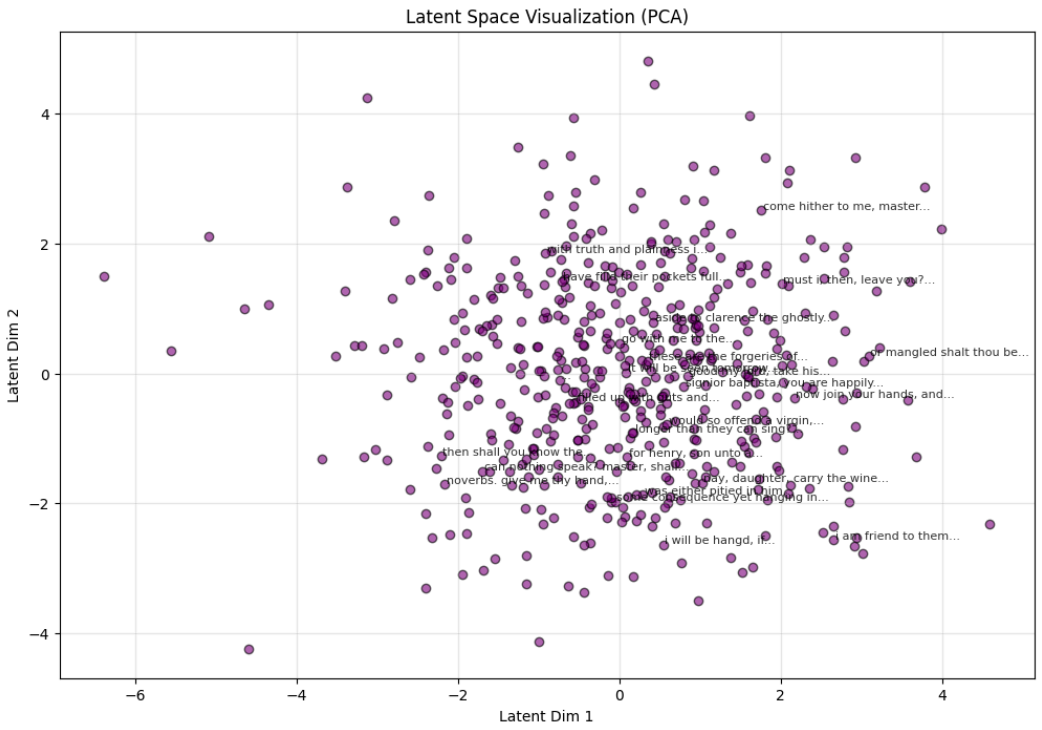
Scatter plot

```
plt.scatter(...)
```

Each point corresponds to a sentence embedding.

📌 PCA Figure

> PCA Projection of Latent Space



> PCA projection of the TextVAE latent space. Each point represents a Shakespeare sentence encoded into the latent space. The dense central distribution and smooth spread indicate effective regularization and meaningful latent representations without collapse.

2.2 t-SNE Visualization (Semantic Clustering)

Thematic prompt groups

```
themes = { Royalty, Love, Conflict }
```

Prompts are grouped by semantic theme to test **latent clustering**.

Encoding prompts

```
z_tensor, _ = encode_text(model, p)
```

Each prompt is encoded into a latent vector.

---

t-SNE reduction

```
TSNE(n_components=2)
```

- Preserves **local neighborhood structure**
- Reveals semantic clusters more clearly than PCA

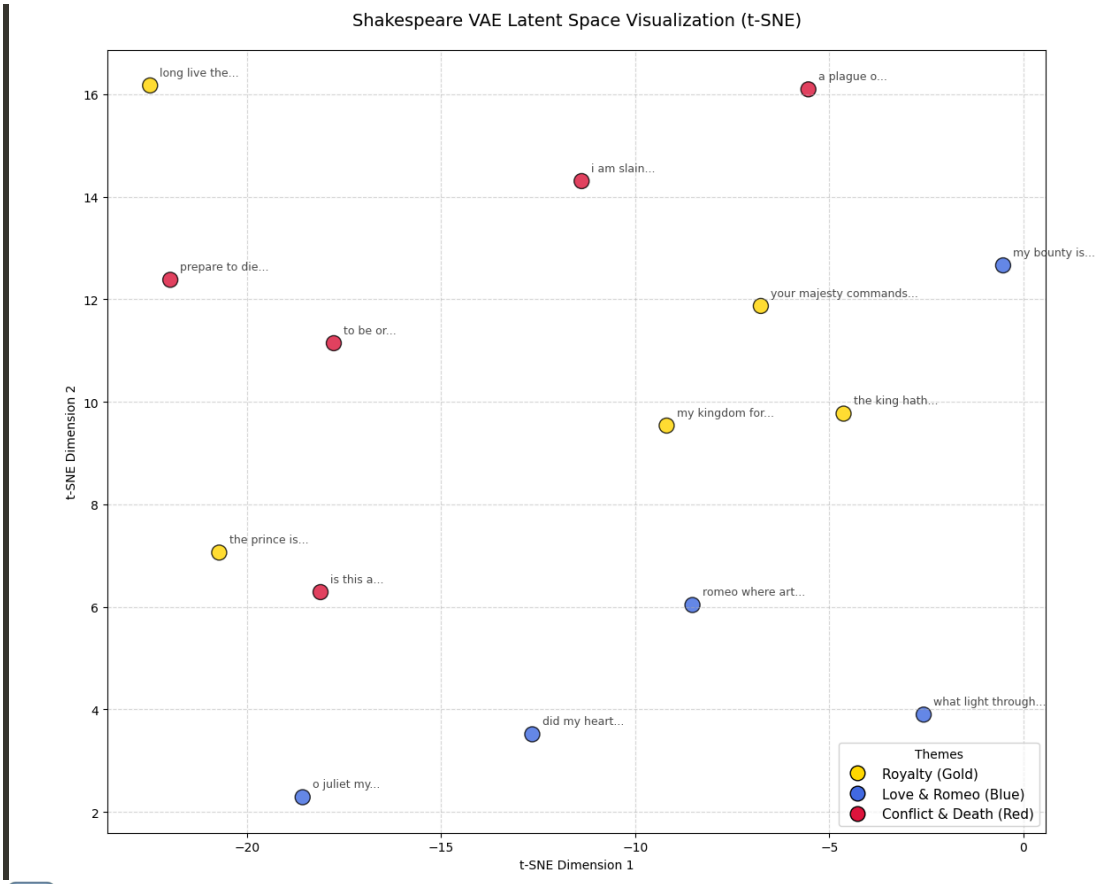
Color-coded scatter plot

```
plt.scatter(..., c=point_colors)
```

Different colors represent different semantic themes.

📌 t-SNE Visualization Placeholder

> t-SNE Visualization of Thematic Latent Clusters



▼ 3️⃣ Text Generation

Objective

To evaluate **fluency, controllability, and diversity** of generated Shakespeare-like text.

🟣 Vocabulary Reconstruction & Token Decoding

Purpose

Neural models operate on **integer token IDs**, but generated outputs must be converted back into **human-readable text**.

Vocabulary Mapping

```
vocab = vectorizer.get_vocabulary()
id_to_token =dict(enumerate(vocab))
```

- `vocab`: Ordered list mapping **token → ID**
- `id_to_token`: Reverse mapping **ID → token**
- Enables decoding predicted token IDs back into words

Token-to-Text Conversion

```
deftokens_to_text(tokens):
    words = [id_to_token.get(t,"")for tin tokensif t >1]
    text =" ".join(words)
return text.replace("start_token","").replace("end_token","").strip()
```

Explanation:

- Ignores:
  - `PAD = 0`
  - `UNK = 1`
- Removes technical training tokens:
  - `start_token`
  - `end_token`
- Produces **clean poetic output** instead of model artifacts

📌 **Why this matters:**

Without cleanup, generated text would expose internal sequence mechanics, harming readability.

🟣 Encoding a Prompt into Latent Space

Convert a text prompt into a **continuous latent representation** that controls generation style and semantics.

Encoding Function

```
defencode_text(model, text):
    text =f"start_token {text} end_token"
    tokens = vectorizer([text])
    enc_in = tokens[:, :-1]
    mean, logvar = model.encode(enc_in)
    z = model.reparameterize(mean, logvar)
return z
```

---

1. **Add boundary tokens**
   - Enables proper sequence learning
2. **Vectorization**

- Converts text → integer IDs

3. **Encoder pass**
   - Produces:
     - `mean` (μ)
     - `logvar` (log σ^2)
4. **Reparameterization**
   - Samples latent vector `z`

📌 **Key insight:**

Each prompt maps to a **region in latent space**, not a fixed sentence, allowing **diverse generations** from the same input.

🟣 Latent Space Interpolation (SLERP)

To verify that the latent space is:

- **Smooth**
- **Continuous**
- **Semantically meaningful**

SLERP Function

```
def slerp(val, low, high):
```

- Performs **spherical linear interpolation**
- Preserves vector magnitude
- Prevents sharp semantic jumps

Why SLERP instead of linear interpolation?

Linear	SLERP
Can collapse semantics	Preserves direction
Cuts through latent space	Moves along manifold
Less stable	Smooth transitions

📌 **Result:**

Generated sentences morph **gradually** from one meaning to another.

🟣 Stabilized Token-by-Token Generation

Core Generation Loop

```
def generate_from_z_stabilized(model, z, max_len=20):
```

This function **decodes text autoregressively**, one token at a time.

---

1. Sequence Initialization

```
current_seq = [start_idx]
generated_ids = []
```

- Starts generation from `start_token`
- Maintains a rolling context window

---

2. Input Preparation

```
seq_input = current_seq[-SEQ_LEN:]
seq_input = [0] * (SEQ_LEN -len(seq_input)) + seq_input
```

- Left-pads sequence
- Maintains fixed input length
- Ensures positional consistency

---

3. Decoding Step

```
logits = model.decode(z, x_input, training=False)
next_token_logits = logits[:, SEQ_LEN-1, :]
```

- Transformer predicts next-token distribution
- Uses **last timestep only**

---

4. Hard Constraints

A. Block PAD and UNK

```
next_token_logits = tf.tensor_scatter_nd_update(
        next_token_logits, [[0, 0], [0, 1]], [-1e9, -1e9]
```

```
        )
```

Prevents meaningless tokens from ever being sampled.

B. Anti-Repetition

```
next_token_logits[last_token] = -1e9
```

Stops immediate word duplication (e.g., *“the the the”*).

---

5. Sampling Strategy

Temperature Scaling

```
logits /0.6
```

- Lower temperature → more deterministic
- Higher temperature → more creative

Top-K Sampling

```
top_k =20
```

- Limits sampling to the 20 most likely tokens
- Prevents rare incoherent words

Random Sampling

```
tf.random.categorical
```

- Introduces controlled randomness
- Enables diversity

---

6. Termination

Stops when:

- `end_token` is sampled
- `max_len` is reached

🟣Advanced Text Completion (Longer Outputs)

The `complete_text()` function is a **production-grade generation pipeline** designed for longer passages.

Improvements Over Basic Generation

Feature	Benefit
Right padding	Accurate time-step prediction
Minimum length	Avoids premature ending
Soft repetition penalty	Prevents looping
Top-P sampling	Adaptive diversity
Temperature control	Creativity tuning

Minimum Length Enforcement

```
if generated_count < min_len:
    block end_token
```

Ensures the model completes a **full poetic thought**.

Anti-Stutter Logic

```
last_token blocked
```

Prevents repeated tokens across adjacent steps.

Soft Repetition Penalty

```
penalize last 5 tokens
```

Reduces probability of recent tokens **without fully blocking them**, preserving natural rhythm.

Top-P (Nucleus) Sampling

```
top_p_sampling(logits, p=0.9)
```

- Selects smallest token set whose cumulative probability ≥ 0.9
- Adapts dynamically to confidence of the model

📌 **Why Top-P over Top-K?**

Top-P automatically adjusts vocabulary size depending on certainty, producing more fluent text.

🟣 Latent Interpolation Sentence Morphing

```
interpolate_sentences()
```

This function:

1. Encodes two prompts
2. Interpolates between their latent vectors
3. Generates text at each step

📌 **What this proves:**

- The model does **not memorize**
- Meaning changes smoothly
- Latent space captures **semantic structure**

🟣 Final Generation Tests

```
test_prompts = [
"to be or not to be",
"the king hath sent",
"what light through yonder",
"romeo where art thou"
]
```

These prompts test:

- Philosophical abstraction
- Authority & command
- Romantic imagery
- Classical Shakespeare motifs

📌 Generated Text

> Generated Text Samples from Different Prompts

```
=== GENERATION TESTS (Longer Sentences) ===

Prompt: to be or not to be
Result: to be or not to be it at your pleasure, as you shall have spoke, and terribly, away by her,

Prompt: the king hath sent
Result: the king hath sent him to the tower, to friar and catesby, his soldier. and, and his hand, another enter the gates of them

Prompt: what light through yonder
Result: what light through yonder rages, and the night his hay. he rages, a comes, and diomedes, terribly, in a captains and maskers

Prompt: romeo where art thou
Result: romeo where art thou that see this? the queen or no? i and thee? the crown guildford suffolk, a soothsayer thomas hastings
```

▼ 4️⃣ Latent Space Interpolation (SLERP)

Objective

To verify that the latent space is **smooth and continuous**.

SLERP Function

```
slerp(alpha, z_start, z_end)
```

- Interpolates between two latent vectors on a hypersphere
- Prevents abrupt semantic jumps

Interpretation

If the latent space is well-structured:

- Generated text changes gradually
- Meaning transitions smoothly

📌 Interpolation Placeholder

> Sentence Morphing via Latent Interpolation

```
Morphing: 'the king hath sent'  ->  'romeo where art thou'

0.0: and here with us to the king he him here to you. they again, and a soldiers
0.2: if it be so, thou must be gone to thee. come, lets away. go away,
0.4: to make a fool of her to my heart. and go with her. i
0.6: to polonius will he know the other sort of it. but i was to him.
0.8: to the of his surveyor, ha? this is a man of soldier. and his
1.0: but wherefore am i entreated her hither, sir offers to the house. and masked then
```

▼ 5️⃣ Diversity vs Coherence Analysis

Objective

To quantify the **fundamental generative trade-off**.

Metrics

Diversity

> Distinct-2 score

Measures bigram uniqueness.

- Calculates the fraction of **unique bigrams (2-word sequences)** in the generated text.
- **High Distinct-2 → high diversity** (less repetition, more variety).
- **Low Distinct-2 → low diversity** (output tends to repeat phrases).

Formula:

$$Distinct - 2 = uniquebigrams/totalbigrams$$

Coherence

> Average probability of sampled tokens

- Measures the **average probability assigned by the model to each chosen token**.
- Acts as a **proxy for confidence**: higher values indicate the model is "more sure" about its choices.
- **High probability → coherent, fluent text**
- **Low probability → risky or less coherent text**

Temperature Sweep

```
temperatures = [0.2,0.5,0.7,1.0,1.2,1.5]
```

- **Temperature** controls randomness during token sampling:
  - Higher temperature → more diversity
  - Lower temperature → more coherence

This sweep allows us to **visualize how adjusting temperature changes the balance between diversity and coherence**.

Methodology

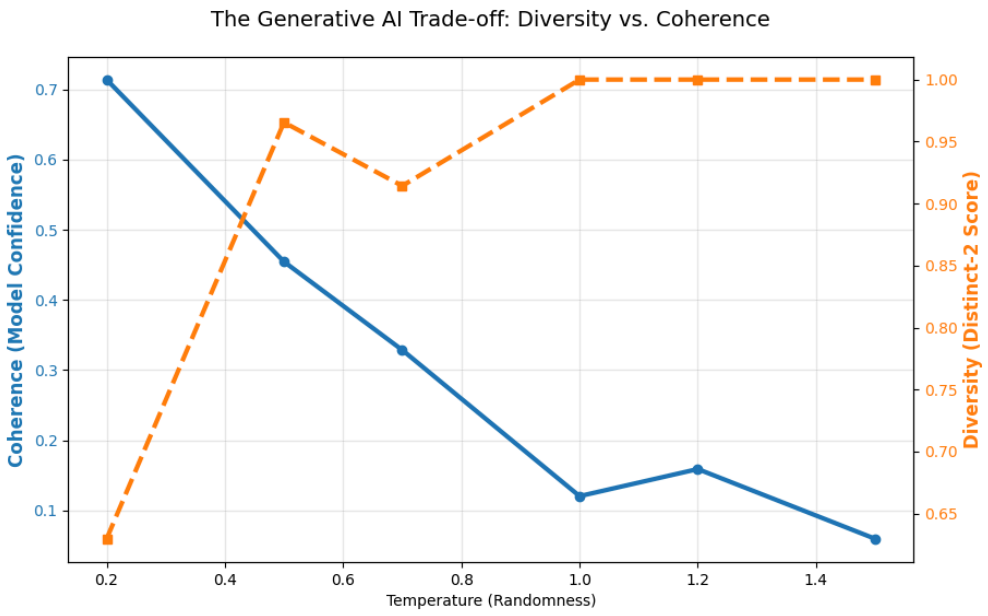
1. **Text Generation per Temperature**
   - For each temperature, generate multiple short sentences from a prompt.
   - Track the probability of each chosen token to compute coherence.

- Collect all generated sentences to compute diversity.

2. **Diversity Calculation**
   - Use `get_distinct_n` function to count unique bigrams.
3. **Coherence Calculation**
   - Compute the average probability of the selected tokens in each sentence.
4. **Aggregation**
   - Compute mean diversity and coherence over all sampled sentences per temperature.
5. **Visualization**
   - Dual-axis plot:
     - Left Y-axis → Coherence (model confidence)
     - Right Y-axis → Diversity (Distinct-2 score)
     - X-axis → Temperature
   - This clearly shows the **trade-off curve**: as diversity increases, coherence generally decreases, and vice versa.

📌 Trade-off Visualization Placeholder

> Diversity vs Coherence Across Sampling Temperatures



▼ 6️⃣ Model Saving & Reproducibility

Saved Components

```
model.save_weights("shakespeare_vae.weights.h5")
```

- Trained model weights

```
vocab.json
```

- Vocabulary mapping (critical for decoding)

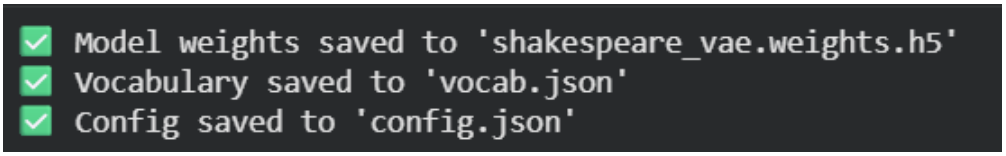
```
config.json
```

- Architecture configuration

Importance

These components ensure:

- Full experiment reproducibility
- Consistent decoding during inference
- Model reusability



▼ 🖥️Interactive GUI for Text Generation

To provide a **user-friendly interface** for interacting with the Shakespeare VAE model. This allows users to **explore text generation, morphing between prompts, and latent space visualizations** without writing any code.

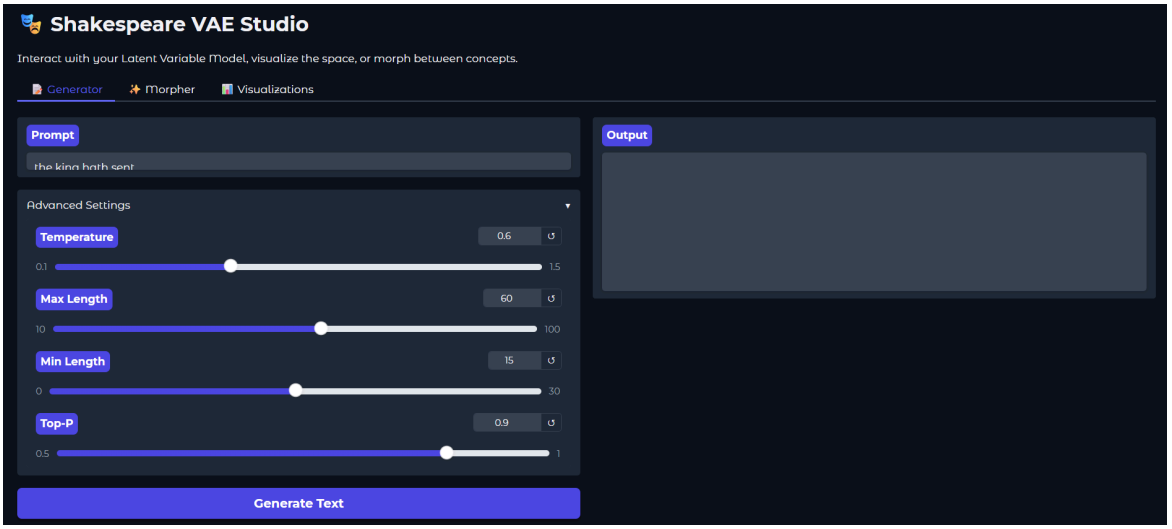
Implementation

- The GUI was created using **Hugging Face Spaces** with **Gradio**.
- It contains three main sections:

1. Generator

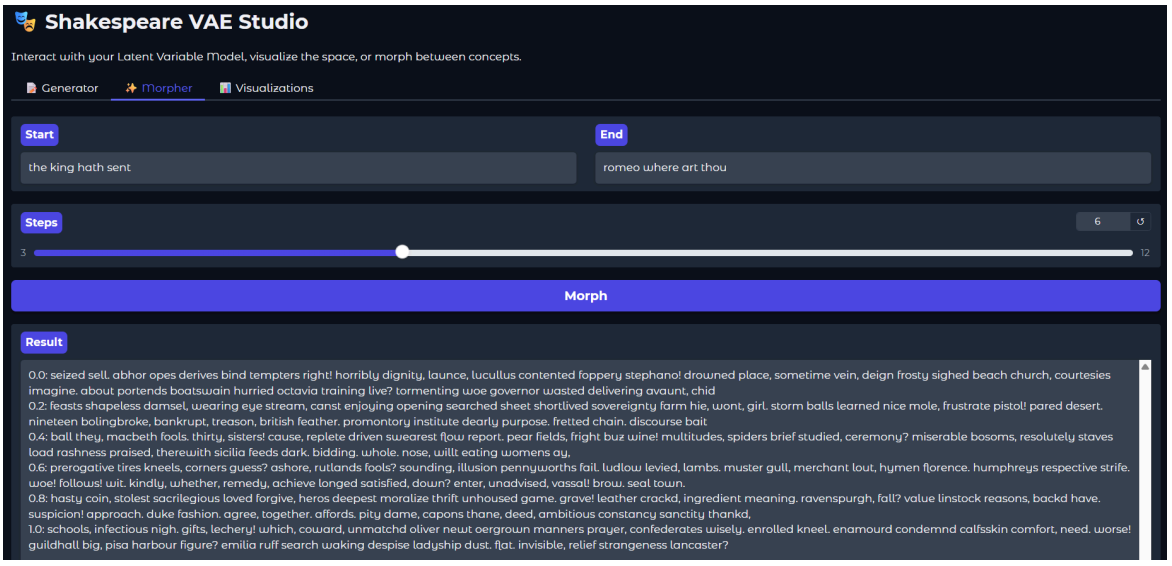
- Allows users to input a **starting text prompt**.
- Users can generate sequences directly from the trained model using **adjustable parameters** such as temperature.

- Outputs coherent Shakespeare-style text based on the trained VAE.



2. Morpher

- Lets users define a **start and end prompt**.
- The model **morphs** from the start text to the end text in a series of intermediate steps.
- Provides insight into how the model transitions across different points in latent space.
- Users can adjust the **number of morphing steps** interactively.

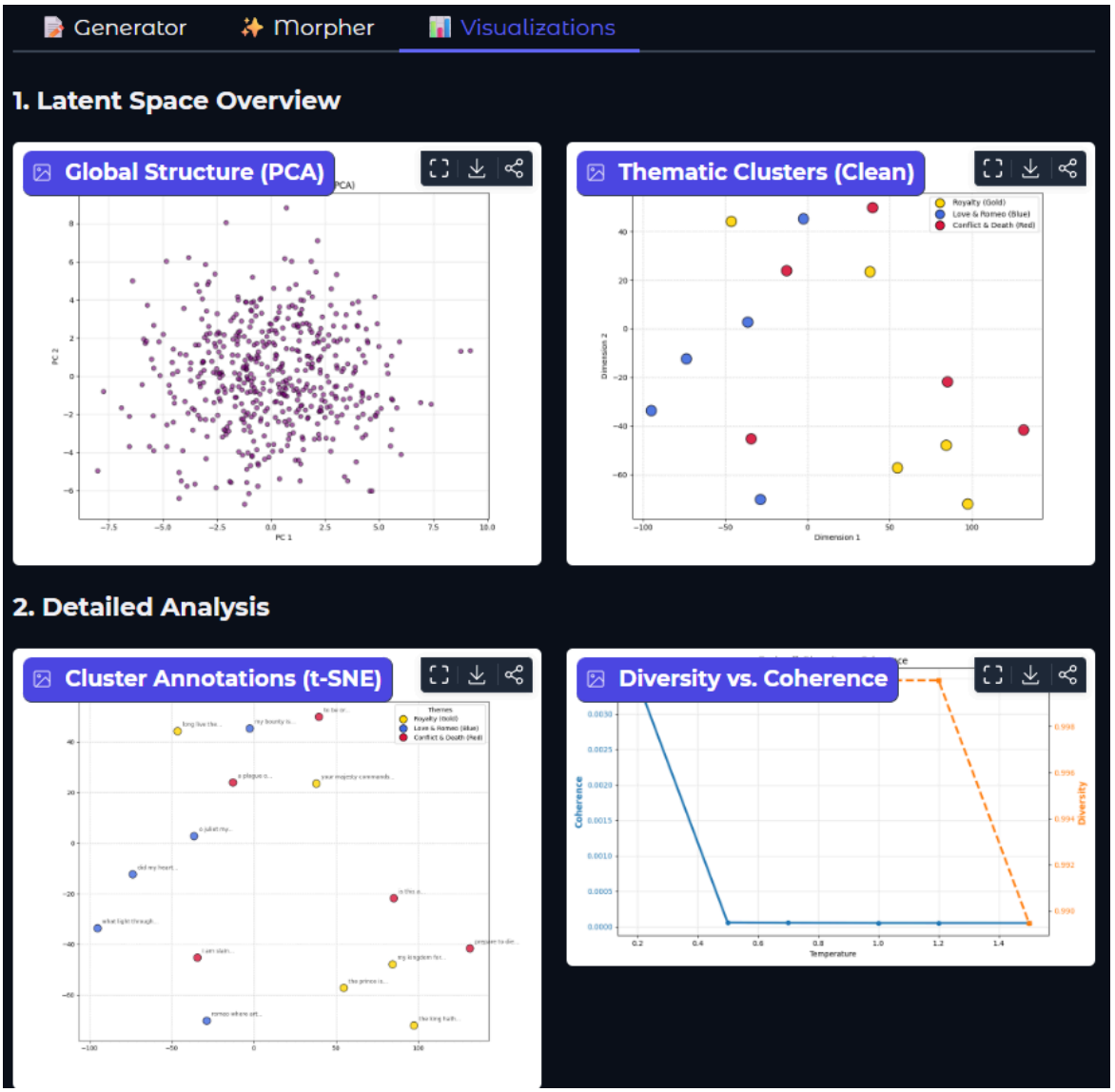


3. Visualizations

- Provides **latent space analysis** of the model:

- **Global Structure (PCA):** Shows overall distribution of embeddings in 2D space.
- **Thematic Clusters:** Highlights clusters of similar themes in the text.
- **Cluster Annotations (t-SNE):** Detailed view of how sentences group by themes.
- Diversity vs. Coherence

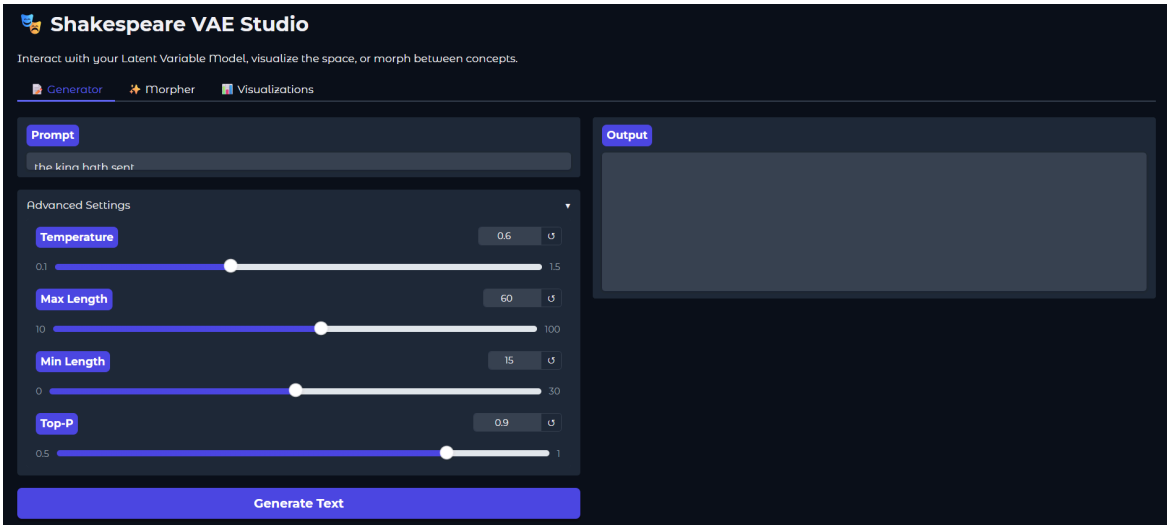
- These visualizations help understand how the VAE organizes text in its latent space.



Access

The live demo is hosted on Hugging Face and can be accessed here:

🔗Text Generation Using VAE – Hugging Face Space



Benefits

1. **No coding required:** Users interact with the model via a simple web interface.
2. **Real-time experimentation:** Adjust temperature and see how diversity and coherence change instantly.
3. **Reproducibility:** Uses the same trained model and vocabulary as in our analysis.

▼ 👨‍💻Source Code

Step 1 : Imports & Setup

```
import tensorflow as tf
import numpy as np
import pandas as pd
import re
import os
import zipfile
from tensorflow.keras import layers
from tqdm.notebook import tqdm
```

Step 2 : GloVe Embeddings Download

```
if not os.path.exists('glove.6B.100d.txt'):
    !wget http://nlp.stanford.edu/data/glove.6B.zip
    !unzip -q glove.6B.zip
        print("Done.")
```

```
else:
    print("GloVe embeddings already present.")
```

Step 3 : Data Preprocessing

```
csv_path = "/content/Shakespeare_data.csv"
txt_path = "/content/alllines.txt"
texts = []

if os.path.exists(csv_path):
    df = pd.read_csv(csv_path)
    texts.extend(df["PlayerLine"].dropna().tolist())

if os.path.exists(txt_path):
    with open(txt_path, "r", encoding="utf-8", errors="ignore") as f:
        texts.extend(f.readlines())

def clean_text(t):
    t = t.lower()
    if ":" in t and len(t.split(":", 1)[0]) < 20:
        t = t.split(":", 1)[1]
    t = re.sub(r"[^a-z0-9\s,.!?]", "", t)
    t = re.sub(r"\s+", " ", t)
    return t.strip()

cleaned_texts = [clean_text(t) for t in texts if len(t.split()) > 5]
unique_texts = list(set(cleaned_texts))
print(f"Total unique samples: {len(unique_texts)}")

formatted_texts = [f"start_token {t} end_token" for t in unique_texts]

VOCAB_SIZE = 20000
SEQ_LEN = 35

vectorizer = layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_mode="int",
    output_sequence_length=SEQ_LEN + 1,
    standardize=None
)
```

```
vectorizer.adapt(formatted_texts)
vocab = vectorizer.get_vocabulary()
```

Step 4 : Dataset Preparation

```
def prepare_batch(text):
    seq = vectorizer(text)
    enc_in = seq[:-1]
    dec_in = seq[:-1]
    target = seq[1:]
    return (enc_in, dec_in), target

dataset = tf.data.Dataset.from_tensor_slices(formatted_texts)
dataset = dataset.map(prepare_batch, num_parallel_calls=tf.data.AUTOTUNE)
dataset = dataset.shuffle(4096).batch(64, drop_remainder=True)
dataset = dataset.prefetch(tf.data.AUTOTUNE)
```

Step 5 : Load GloVe Embeddings

```
def load_glove_embeddings(vocab, embed_dim=100):
    embeddings_index = {}
    with open(f"glove.6B.{embed_dim}d.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
    num_tokens = len(vocab)
    embedding_matrix = np.zeros((num_tokens, embed_dim))
    hits = 0
    misses = 0
    for i, word in enumerate(vocab_list):
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
        hits += 1
        else:
        misses += 1

        print(f"Converted {hits} words ({misses} misses)")
        return embedding_matrix
```

```
EMBED_DIM = 100
embedding_matrix = load_glove_embeddings(vocab, EMBED_DIM)
```

Step 6 : Positional Encoding

```
def positional_encoding(seq_len, d_model):
    pos = np.arange(seq_len)[:, None]
    i = np.arange(d_model)[None, :].astype(np.float32)
    angle_rates = 1 / np.power(10000, (2 * (i // 2)) / d_model)
    angle_rads = pos * angle_rates
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    return tf.cast(angle_rads[None, ...], tf.float32)
```

Step 7 : VAE + Transformer Model

```
class TextVAE(tf.keras.Model):
    def __init__(self, vocab_size, seq_len, latent_dim, embed_dim, embedding_matr
ix):
        super().__init__()
        self.latent_dim = latent_dim
        self.seq_len = seq_len
        self.embed_dim = embed_dim

        self.embedding = layers.Embedding(
            vocab_size,
            embed_dim,
            embeddings_initializer=tf.keras.initializers.Constant(embedding_matrix),
            trainable=True
        )

        self.encoder_lstm = layers.Bidirectional(layers.LSTM(256))
        self.encoder_dense = layers.Dense(latent_dim * 2)

        self.latent_projection = layers.Dense(seq_len * embed_dim)
        self.pos_encoding = positional_encoding(seq_len, embed_dim)

        self.decoder_layers = [
            layers.MultiHeadAttention(num_heads=4, key_dim=embed_dim) for _ in ra
nge(2)
```

```
        ]
        self.ffn_layers = [
            tf.keras.Sequential([
                layers.Dense(512, activation="relu"),
                layers.Dense(embed_dim)
            ]) for _ in range(2)
        ]
        self.layernorm1 = [layers.LayerNormalization(epsilon=1e-6) for _ in range(2)]
        self.layernorm2 = [layers.LayerNormalization(epsilon=1e-6) for _ in range
(2)]

        self.output_layer = layers.Dense(vocab_size)

    def encode(self, x):
        x = self.embedding(x)
        x = self.encoder_lstm(x)
        mean, logvar = tf.split(self.encoder_dense(x), 2, axis=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=tf.shape(mean))
        return eps * tf.exp(0.5 * logvar) + mean

    def decode(self, z, x, training=False, word_dropout_rate=0.0):
        if training and word_dropout_rate > 0:
            rand = tf.random.uniform(shape=tf.shape(x))
            drop_mask = tf.cast(rand > word_dropout_rate, x.dtype)
            x = x * drop_mask + 1 * (1 - drop_mask) # 1 is [UNK]

        x_emb = self.embedding(x)
        x_emb += self.pos_encoding

        z_proj = self.latent_projection(z)
        z_proj = tf.reshape(z_proj, (-1, self.seq_len, self.embed_dim))
        h = x_emb + z_proj

        mask = tf.linalg.band_part(tf.ones((self.seq_len, self.seq_len)), -1, 0)

        for i in range(len(self.decoder_layers)):
            attn_out = self.decoder_layers[i](h, h, attention_mask=mask)
            h = self.layernorm1[i](h + attn_out)
            ffn_out = self.ffn_layers[i](h)
            h = self.layernorm2[i](h + ffn_out)
```

```
        return self.output_layer(h)

latent_dim = 128
MODEL_SEQ_LEN = SEQ_LEN
model = TextVAE(VOCAB_SIZE, MODEL_SEQ_LEN, latent_dim, EMBED_DIM, emb
edding_matrix)
```

Step 8 : Optimizer & Learning Rate

```
try:
    steps_per_epoch = len(dataset)
    if steps_per_epoch == 0:
        print("⚠️ Warning: Dataset length is 0. Defaulting to 1000 steps.")
        steps_per_epoch = 1000
except:
    print("⚠️ Warning: Dataset not found. Defaulting to 1000 steps.")
    steps_per_epoch = 1000

decay_steps = 30 * steps_per_epoch

lr_schedule = tf.keras.optimizers.schedules.CosineDecay(
    initial_learning_rate=0.001,
    decay_steps=decay_steps,
    alpha=0.1
)

optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule, clipnorm=1.0)
print("✅ Optimizer ready.")
```

Step 9 : Training Loop

```
lr_schedule = tf.keras.optimizers.schedules.CosineDecay(
    initial_learning_rate=0.001,
    decay_steps=30 * len(dataset),
    alpha=0.1
)

optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule, clipnorm=1.0)

EPOCHS = 30
```

```
KL_WARMUP = 5

@tf.function
def train_step(inputs, target, beta, word_dropout):
    enc_in, dec_in = inputs

    with tf.GradientTape() as tape:
        mean, logvar = model.encode(enc_in)
        z = model.reparameterize(mean, logvar)

        logits = model.decode(z, dec_in, training=True, word_dropout_rate=word_dr
opout)

        recon_loss = tf.reduce_mean(
            tf.keras.losses.sparse_categorical_crossentropy(
                target, logits, from_logits=True
            )
        )
        kl_loss = -0.5 * tf.reduce_mean(
            tf.reduce_sum(1 + logvar - tf.square(mean) - tf.exp(logvar), axis=1)
        )
        kl_loss_clipped = tf.maximum(kl_loss, 0.1)

        total_loss = recon_loss + (beta * kl_loss_clipped)

    grads = tape.gradient(total_loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return total_loss, recon_loss, kl_loss

    print(f"Starting Training on {len(dataset)} batches...")

for epoch in range(EPOCHS):
    if epoch < KL_WARMUP:
        BETA = (epoch / KL_WARMUP) * 0.5
    else:
        BETA = 0.5

    epoch_loss = 0.0
    epoch_recon = 0.0
    epoch_kl = 0.0
    steps = 0

    progress = tqdm(dataset, desc=f"Epoch {epoch+1}", leave=False)
```

```
    for batch in progress:
        (enc_in, dec_in), target = batch
        loss, recon, kl = train_step((enc_in, dec_in), target, BETA, word_dropout=0.
3)

        epoch_loss += loss
        epoch_recon += recon
        epoch_kl += kl
        steps += 1

        progress.set_postfix({
            "loss": f"{loss:.3f}",
            "recon": f"{recon:.3f}",
            "kl": f"{kl:.3f}"
        })

    avg_loss = epoch_loss / steps if steps > 0 else 0
    avg_recon = epoch_recon / steps if steps > 0 else 0
    avg_kl = epoch_kl / steps if steps > 0 else 0

    print(f"Epoch {epoch+1} | Loss: {avg_loss:.4f} | Recon: {avg_recon:.4f} | KL: {a
vg_kl:.4f}")

print("Training Complete!")
```

Step 10 : Evaluation & Latent Space

```
from sklearn.metrics import accuracy_score, precision_score, recall_score
import numpy as np

def evaluate_model(model, dataset, max_batches=50):
    y_true = []
    y_pred = []

    print(f"Evaluating on {max_batches} batches...")

    for i, batch in enumerate(dataset):
        if i >= max_batches:
            break

        (enc_in, dec_in), target = batch
```

```
        mean, logvar = model.encode(enc_in)
        z = model.reparameterize(mean, logvar)

        logits = model.decode(z, dec_in, training=False)

        predictions = tf.argmax(logits, axis=-1)

        y_true.extend(target.numpy().flatten())
        y_pred.extend(predictions.numpy().flatten())

    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    mask = y_true != 0
    y_true_filtered = y_true[mask]
    y_pred_filtered = y_pred[mask]

    acc = accuracy_score(y_true_filtered, y_pred_filtered)

    prec = precision_score(y_true_filtered, y_pred_filtered, average='macro', zero_
division=0)
    rec = recall_score(y_true_filtered, y_pred_filtered, average='macro', zero_divisi
on=0)

    print("\n=== Model Performance ===")
    print(f"Token-Level Accuracy:  {acc:.4f}")
    print(f"Macro Precision:       {prec:.4f}")
    print(f"Macro Recall:          {rec:.4f}")

    return y_true_filtered, y_pred_filtered

y_true_eval, y_pred_eval = evaluate_model(model, dataset)
```

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

subset_texts = unique_texts[:500]
subset_vecs = vectorizer(np.array(subset_texts))

mean, logvar = model.encode(subset_vecs)
z_subset = model.reparameterize(mean, logvar)
```

```
pca = PCA(n_components=2)
z_2d = pca.fit_transform(z_subset.numpy())

plt.figure(figsize=(12, 8))
plt.scatter(z_2d[:, 0], z_2d[:, 1], alpha=0.6, c='purple', edgecolors='k')

for i in range(0, 500, 20):
    text_snippet = " ".join(subset_texts[i].split()[:5]) + "..."
    plt.text(z_2d[i, 0]+0.02, z_2d[i, 1]+0.02, text_snippet, fontsize=8, alpha=0.8)

plt.title("Latent Space Visualization (PCA)")
plt.xlabel("Latent Dim 1")
plt.ylabel("Latent Dim 2")
plt.grid(True, alpha=0.3)
plt.show()
```

Step 11 : Text Generation & Interpolation

```
vocab = vectorizer.get_vocabulary()
id_to_token = dict(enumerate(vocab))

def tokens_to_text(tokens):
    words = [id_to_token.get(t, "") for t in tokens if t > 1]
    text = " ".join(words)
    text = text.replace("start_token", "").replace("end_token", "")
    return text.strip()

def encode_text(model, text):
    text = f"start_token {text} end_token"
    tokens = vectorizer([text])
    enc_in = tokens[:, :-1]
    mean, logvar = model.encode(enc_in)
    z = model.reparameterize(mean, logvar)
    return z

def slerp(val, low, high):
    low_norm = low / tf.norm(low)
    high_norm = high / tf.norm(high)
    omega = tf.acos(tf.clip_by_value(tf.tensordot(low_norm, high_norm, axes=1), -
1, 1))
    so = tf.sin(omega)
```

```
    if so == 0: return (1.0 - val) * low + val * high
    return (tf.sin((1.0 - val) * omega) / so) * low + (tf.sin(val * omega) / so) * high

def generate_from_z_stabilized(model, z, max_len=20):
    try:
        start_idx = list(vocab).index("start_token")
    except:
        start_idx = 1

    current_seq = [start_idx]
    generated_ids = []

    for _ in range(max_len):
        seq_input = current_seq[-SEQ_LEN:]
        if len(seq_input) < SEQ_LEN:
            seq_input = [0] * (SEQ_LEN - len(seq_input)) + seq_input

        x_input = tf.convert_to_tensor([seq_input], dtype=tf.int64)

        logits = model.decode(z, x_input, training=False)
        next_token_logits = logits[:, SEQ_LEN-1, :]

        next_token_logits = tf.tensor_scatter_nd_update(
            next_token_logits, [[0, 0], [0, 1]], [-1e9, -1e9]
        )

        if len(generated_ids) > 0:
            last_token = generated_ids[-1]
            next_token_logits = tf.tensor_scatter_nd_update(
                next_token_logits, [[0, last_token]], [-1e9]
            )

        next_token_logits = next_token_logits / 0.6
        values, indices = tf.math.top_k(next_token_logits, k=20)

        sampled_index = tf.random.categorical(values, num_samples=1)[0, 0]
        next_token_id = indices[0, sampled_index].numpy()

        try:
            if next_token_id == list(vocab).index("end_token"):
                break
        except:
            pass
```

```
        generated_ids.append(next_token_id)
        current_seq.append(next_token_id)

    return tokens_to_text(generated_ids)

def interpolate_sentences(model, start_text, end_text, steps=6):
    z_start = tf.squeeze(encode_text(model, start_text))
    z_end = tf.squeeze(encode_text(model, end_text))

    alphas = np.linspace(0, 1, steps)
    print(f"Morphing (SLERP): '{start_text}'  →  '{end_text}'\n")

    for alpha in alphas:
        z_interp = slerp(alpha, z_start, z_end)
        z_interp = tf.expand_dims(z_interp, 0)
        text = generate_from_z_stabilized(model, z_interp)
        print(f"{alpha:.1f}: {text}")
```

Step 12 : Run Text Generation Tests

```
vocab = vectorizer.get_vocabulary()
id_to_token = dict(enumerate(vocab))

def tokens_to_text(tokens):
    words = [id_to_token.get(t, "") for t in tokens if t > 1]
    text = " ".join(words)
    return text.replace("start_token", "").replace("end_token", "").strip()

def encode_text(model, text):
    text = f"start_token {text} end_token"
    tokens = vectorizer([text])
    enc_in = tokens[:, :-1]
    mean, logvar = model.encode(enc_in)
    z = model.reparameterize(mean, logvar)
    return z, tokens

def top_p_sampling(logits, p=0.9):
    probs = tf.nn.softmax(logits)
    sorted_probs = tf.sort(probs, direction='DESCENDING')
    cumulative_probs = tf.math.cumsum(sorted_probs, axis=-1)
```

```
    cutoff_idx = tf.reduce_sum(tf.cast(cumulative_probs < p, tf.int32))
    cutoff_idx = tf.minimum(cutoff_idx, tf.shape(logits)[-1] - 1)
    cutoff_prob = tf.gather(sorted_probs, cutoff_idx, axis=-1)

    return tf.where(probs < cutoff_prob, tf.fill(tf.shape(logits), -1e9), logits)

def complete_text(model, prompt, max_new_tokens=60, min_len=15, temperature
=0.7, top_p=0.9, seq_len=SEQ_LEN):
    z, tokens_tensor = encode_text(model, prompt)
    tokens = tokens_tensor.numpy()[0].tolist()
    tokens = [t for t in tokens if t != 0] # Remove padding

    try:
        end_id = list(vocab).index("end_token")
        if tokens and tokens[-1] == end_id:
            tokens.pop()
    except ValueError:
        end_id = 0

    generated_count = 0

    for _ in range(max_new_tokens):
        current_seq = tokens[-seq_len:]
        if len(current_seq) < seq_len:
            current_seq = current_seq + [0] * (seq_len - len(current_seq))

        x_input = tf.convert_to_tensor([current_seq], dtype=tf.int64)

        logits = model.decode(z, x_input, training=False)

        valid_len = len([t for t in current_seq if t != 0])
        pred_idx = min(valid_len - 1, seq_len - 1)
        next_logits = logits[:, pred_idx, :] / temperature

        if generated_count < min_len:
            next_logits = tf.tensor_scatter_nd_update(
                next_logits, [[0, end_id], [0, 0]], [-1e9, -1e9]
            )
        else:
            next_logits = tf.tensor_scatter_nd_update(
                next_logits, [[0, 0]], [-1e9]
            )
```

```
        last_token = tokens[-1]
        next_logits = tf.tensor_scatter_nd_update(
            next_logits, [[0, last_token]], [-1e9]
        )

        recent_tokens = set(tokens[-5:])
        for t_id in recent_tokens:
            if t_id != last_token:
                next_logits = tf.tensor_scatter_nd_sub(
                    next_logits, [[0, t_id]], [2.0]
                )

        next_logits = top_p_sampling(next_logits, p=top_p)

        next_token_id = tf.random.categorical(next_logits, num_samples=1)[0, 0].nu
mpy()

        if next_token_id == end_id:
            break

        tokens.append(next_token_id)
        generated_count += 1

    return tokens_to_text(tokens)

test_prompts = [
    "to be or not to be",
    "the king hath sent",
    "what light through yonder",
    "romeo where art thou"
]

print("\n=== GENERATION TESTS (Longer Sentences) ===")
for p in test_prompts:
    generated = complete_text(model, p, max_new_tokens=60, min_len=15, tempe
rature=0.8)
    print(f"\nPrompt: {p}")
    print(f"Result: {generated}")
```

Step 13 : Latent Space Morphing

```
def slerp(val, low, high):
    low_norm = low / tf.norm(low)
    high_norm = high / tf.norm(high)
    omega = tf.acos(tf.clip_by_value(tf.tensordot(low_norm, high_norm, axes=1), -
1, 1))
    so = tf.sin(omega)
    if so == 0: return (1.0 - val) * low + val * high
    return (tf.sin((1.0 - val) * omega) / so) * low + (tf.sin(val * omega) / so) * high

def generate_from_z(model, z, max_len=60, min_len=15, temperature=0.7, top_p
=0.9):
    try:
        start_id = list(vocab).index("start_token")
        end_id = list(vocab).index("end_token")
    except:
        start_id = 1
        end_id = 0

    current_seq = [start_id]
    generated_ids = []

    for _ in range(max_len):
        seq_input = current_seq[-SEQ_LEN:]
        if len(seq_input) < SEQ_LEN:
            seq_input = seq_input + [0] * (SEQ_LEN - len(seq_input))

        x_input = tf.convert_to_tensor([seq_input], dtype=tf.int64)

        logits = model.decode(z, x_input, training=False)

        valid_len = len([t for t in current_seq if t != 0])
        pred_idx = min(valid_len - 1, SEQ_LEN - 1)
        next_logits = logits[:, pred_idx, :] / temperature

        if len(generated_ids) < min_len:
            next_logits = tf.tensor_scatter_nd_update(
                next_logits, [[0, end_id], [0, 0]], [-1e9, -1e9]
            )
        else:
            next_logits = tf.tensor_scatter_nd_update(
                next_logits, [[0, 0]], [-1e9]
            )
```

```
        if generated_ids:
            last_token = generated_ids[-1]
            next_logits = tf.tensor_scatter_nd_update(
                next_logits, [[0, last_token]], [-1e9]
            )

        recent = set(generated_ids[-5:])
        for t in recent:
            if generated_ids and t != generated_ids[-1]:
                next_logits = tf.tensor_scatter_nd_sub(next_logits, [[0, t]], [2.0])

        next_logits = top_p_sampling(next_logits, p=top_p)

        next_token_id = tf.random.categorical(next_logits, num_samples=1)[0, 0].nu
mpy()

        if next_token_id == end_id:
            break

        generated_ids.append(next_token_id)
        current_seq.append(next_token_id)

    return tokens_to_text(generated_ids)

def interpolate_sentences(model, start_text, end_text, steps=6):
    z_start, _ = encode_text(model, start_text)
    z_end, _   = encode_text(model, end_text)

    z_start = tf.squeeze(z_start)
    z_end = tf.squeeze(z_end)

    alphas = np.linspace(0, 1, steps)

    print(f"Morphing: '{start_text}'  →  '{end_text}'\n")

    for alpha in alphas:
        z_interp = slerp(alpha, z_start, z_end)
        z_interp = tf.expand_dims(z_interp, 0)

        text = generate_from_z(model, z_interp, max_len=60, min_len=15)
        print(f"{alpha:.1f}: {text}")
```

```
interpolate_sentences(model, "the king hath sent", "romeo where art thou", steps
=6)
```

Step 14 — t-SNE Latent Visualization

```
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import numpy as np
from matplotlib.lines import Line2D

themes = {
    "Royalty (Gold)": [
        "the king hath sent",
        "my kingdom for a horse",
        "long live the king",
        "your majesty commands it",
        "the prince is angry"
    ],
    "Love & Romeo (Blue)": [
        "romeo where art thou",
        "what light through yonder window breaks",
        "did my heart love till now",
        "o juliet my sweet",
        "my bounty is as boundless as the sea"
    ],
    "Conflict & Death (Red)": [
        "to be or not to be",
        "a plague o both your houses",
        "is this a dagger which i see",
        "i am slain",
        "prepare to die"
    ]
}

z_vectors = []
point_labels = []
point_colors = []
theme_color_map = {
    "Royalty (Gold)": "gold",
    "Love & Romeo (Blue)": "royalblue",
    "Conflict & Death (Red)": "crimson"
}
```

```
print("Encoding prompts into 128D latent space...")
for theme, prompts in themes.items():
    for p in prompts:
        z_tensor, _ = encode_text(model, p)
        z_vec = z_tensor.numpy().flatten()

        z_vectors.append(z_vec)
        point_labels.append(p)
        point_colors.append(theme_color_map[theme])

z_matrix = np.array(z_vectors)

print("Running t-SNE reduction (128D → 2D)...")
tsne = TSNE(n_components=2, perplexity=5, random_state=42, n_iter=2000, init
='pca')
z_2d = tsne.fit_transform(z_matrix)

plt.figure(figsize=(12, 10))

plt.scatter(z_2d[:, 0], z_2d[:, 1], c=point_colors, s=150, edgecolors='k', alpha=0.
8)

for i, label in enumerate(point_labels):
    short_label = " ".join(label.split()[:3]) + "..."
    plt.annotate(short_label, (z_2d[i, 0], z_2d[i, 1]),
                 xytext=(8, 8), textcoords='offset points', fontsize=9, alpha=0.7)

legend_elements = [Line2D([0], [0], marker='o', color='w', label=k,
                          markerfacecolor=v, markersize=12, markeredgecolor='k')
                   for k, v in theme_color_map.items()]
plt.legend(handles=legend_elements, loc='best', title="Themes", fontsize=11)

plt.title("Shakespeare VAE Latent Space Visualization (t-SNE)", fontsize=14, pad=
20)
plt.xlabel("t-SNE Dimension 1")
plt.ylabel("t-SNE Dimension 2")
plt.grid(True, linestyle='--', alpha=0.5)
plt.tight_layout()

print("Plot generated successfully.")
plt.show()
```

Step 15 : Diversity and Coherence Analysis

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

def get_distinct_n(corpus, n=2):
    if not corpus: return 0.0
    bigrams = []
    for sentence in corpus:
        tokens = sentence.split()
        if len(tokens) < n: continue
        for i in range(len(tokens) - n + 1):
            ngram = tuple(tokens[i:i+n])
            bigrams.append(ngram)
    if len(bigrams) == 0: return 0.0
    return len(set(bigrams)) / len(bigrams)

def analyze_generation(model, prompt, temps):
    diversity_scores = []
    coherence_scores = []
    z, _ = encode_text(model, prompt)
    print(f"Analyzing prompt: '{prompt}'...")
    for temp in temps:
        generated_sentences = []
        token_probs = []
        for _ in range(5):
            current_seq = [list(vocab).index("start_token")]
            seq_probs = []
            for _ in range(20): # Short length for speed
                seq_in = current_seq[-SEQ_LEN:]
                if len(seq_in) < SEQ_LEN: seq_in = seq_in + [0]*(SEQ_LEN - len(seq_i
n))
                x_in = tf.convert_to_tensor([seq_in], dtype=tf.int64)

                logits = model.decode(z, x_in, training=False)
                next_logits = logits[:, min(len(current_seq)-1, SEQ_LEN-1), :] / temp

                probs = tf.nn.softmax(next_logits)

                next_id = tf.random.categorical(next_logits, num_samples=1)[0, 0].num
py()
```

```
            chosen_prob = probs[0, next_id].numpy()
            seq_probs.append(chosen_prob)

            if next_id == list(vocab).index("end_token"): break
            current_seq.append(next_id)

        text = tokens_to_text(current_seq)
        generated_sentences.append(text)
        token_probs.append(np.mean(seq_probs))

    div_score = get_distinct_n(generated_sentences, n=2)
    coh_score = np.mean(token_probs)

    diversity_scores.append(div_score)
    coherence_scores.append(coh_score)
    print(f"  Temp {temp:.1f} | Diversity: {div_score:.2f} | Confidence: {coh_scor
e:.2f}")

  return diversity_scores, coherence_scores

temperatures = [0.2, 0.5, 0.7, 1.0, 1.2, 1.5]
prompt = "the king hath sent"

divs, cohs = analyze_generation(model, prompt, temperatures)

fig, ax1 = plt.subplots(figsize=(10, 6))

color = 'tab:blue'
ax1.set_xlabel('Temperature (Randomness)')
ax1.set_ylabel('Coherence (Model Confidence)', color=color, fontsize=12, fontwei
ght='bold')
ax1.plot(temperatures, cohs, color=color, marker='o', linewidth=3, label='Cohere
nce')
ax1.tick_params(axis='y', labelcolor=color)
ax1.grid(True, alpha=0.3)

ax2 = ax1.twinx()
color = 'tab:orange'
ax2.set_ylabel('Diversity (Distinct-2 Score)', color=color, fontsize=12, fontweight
='bold')
ax2.plot(temperatures, divs, color=color, marker='s', linestyle='--', linewidth=3, l
abel='Diversity')
ax2.tick_params(axis='y', labelcolor=color)
```

```
plt.title('The Generative AI Trade-off: Diversity vs. Coherence', fontsize=14, pad=
20)
plt.show()
```

Step 16 : Model and Vocabulary Saving

```
import json
import numpy as np

model.build(input_shape=(None, SEQ_LEN))
model.save_weights("shakespeare_vae.weights.h5")
print("✅ Model weights saved to 'shakespeare_vae.weights.h5'")

vocab_list = vectorizer.get_vocabulary()

with open("vocab.json", "w") as f:
    json.dump(vocab_list, f)
print("✅ Vocabulary saved to 'vocab.json'")

config = {
    "vocab_size": len(vocab_list),
    "seq_len": SEQ_LEN,
    "latent_dim": 32,
    "embed_dim": 256,
    "ff_dim": 512
}
with open("config.json", "w") as f:
    json.dump(config, f)
print("✅ Config saved to 'config.json'")
```

▼ Conclusion

🎯 This project demonstrates the use of a **Variational Autoencoder (VAE)** for generating Shakespearean-style text. By experimenting with different **temperature settings**, we observed how the model balances **creativity (diversity)** and **coherence (confidence in word choice)**.
We also implemented **evaluation metrics** to quantify these aspects, providing insights into how temperature and sampling strategies impact text quality. The project includes a complete **pipeline with trained model weights, vocabulary, and configuration files**, making it reproducible and easy to extend.