Portfolio of Evidence

Part 1– RNN and LTSM with text

Stylometry / Authorship RNN

Full Methodology & Report

Rick Jackson

ST10458509

3 October 2025

Varsity College Cape Town | Emeris Cape Town

Table of Contents

Introduction ***3***

Dataset & Ingestion ***3***

Preprocessing & Tokenization (code as provided) ***3***

Original Tokenization *3*

Stemming (PorterStemmer) *4*

Lemmatization (WordNetLemmatizer) *5*

Vocabulary comparison and sequence-length distributions *6*

Word Clouds (EDA visuals) ***6***

Model architecture & training ***6***

Results (as recorded in the notebook) ***8***

Original tokens — classification report (validation) *8*

Stemmed tokens — classification report (validation) *8*

Lemmatized tokens — classification report (validation) *8*

Aggregated results table (notebook results_df) 8

Visual comparisons (Accuracy and F1-score) ***9***

Discussion & Interpretation ***9***

Conclusion ***9***

Appendix A — Full code (concise) ***10***

This report documents an end-to-end experiment in **authorship attribution (stylometry)** using the Spooky Author dataset. The pipeline explores three preprocessing variants — **original tokens**, **Porter stemming**, and **WordNet lemmatization** — and trains a baseline neural model on each. The goal is to compare how preprocessing affects vocabulary, sequence structure, and classification performance.

This document includes: rationale, all code blocks exactly as executed, experimental results (vocabulary sizes, sample sequences, training logs), visual comparisons (accuracy & F1 charts), and interpretation of findings.

Introduction

Stylometry aims to identify the author of a piece of text by quantifying writing style. In many real-world contexts (forensics, literary studies, plagiarism detection), subtle differences in word choice, punctuation, and phrasing can carry strong signals. This project investigates how **text preprocessing choices** (no preprocessing, stemming, lemmatization) impact model performance.

Rationale for model choice: - The model must handle sequential text features; recurrent or embedding-based approaches are typical. - For reproducibility and simplicity, a baseline neural classifier is trained on tokenized sequences (the code used a simple dense/embedding-based model architecture for quick experiments).

Dataset & Ingestion

Dataset used: Spooky Author dataset (three authors: EAP, HPL, MWS). The dataset was loaded using Spark and carefully parsed with multiLine=True and escape="" to avoid CSV parsing issues. After cleaning, the dataset was sampled to ~10,000 rows for experiments.

Preprocessing & Tokenization

Below are the code blocks exactly as used in the experiment. Each block is followed by short notes explaining the purpose and the observed outputs.

Original Tokenization

```
# Tokenizer settings
MAX_WORDS = 10000   # max vocab size
MAX_LEN = 150       # max sequence length (based on EDA text length)
```

```
tokenizer = Tokenizer(num_words=MAX_WORDS, oov_token="<OOV>")
tokenizer.fit_on_texts(df_pandas['text'])

# Conversion of text to sequences
sequences = tokenizer.texts_to_sequences(df_pandas['text'])

# Pad sequences
X = pad_sequences(sequences, maxlen=MAX_LEN, padding='post', truncating='post')

# Labels
y = df_pandas['author_id'].values
```

Purpose: Build a fixed vocabulary (top MAX_WORDS) and convert text to fixed-length integer sequences for model input.

Observed outputs (original): - Vocabulary size: 19,903 unique tokens - Example token mapping & padded sequence printed in the notebook - Average sequence length: ~26 tokens (min 2, max 256)

Stemming (PorterStemmer)

```
# Convert DataFrame column to a list of strings
texts = df_pandas['text'].tolist()

# Tokenize each text
tokenized_texts = [word_tokenize(text) for text in texts]

# Apply stemming
stemmer = PorterStemmer()
stemmed_texts = [[stemmer.stem(word) for word in tokens] for tokens in tokenized_texts]

# Join tokens for Keras Tokenizer
stemmed_texts_joined = [' '.join(tokens) for tokens in stemmed_texts]

# Tokenizer settings
MAX_WORDS = 10000
MAX_LEN = 150

tokenizer_stem = Tokenizer(num_words=MAX_WORDS, oov_token="<OOV>")
tokenizer_stem.fit_on_texts(stemmed_texts_joined)

# Convert to sequences
X_stem = tokenizer_stem.texts_to_sequences(stemmed_texts_joined)
```

```
# Pad sequences
X_stem = pad_sequences(X_stem, maxlen=MAX_LEN, padding='post', truncating='post')

# Labels
y = df_pandas['author_id'].values
```

Purpose: Reduce vocabulary size by stemming, potentially reducing sparsity and improving classifier generalization.

Observed outputs (stemming): - Vocabulary size after stemming: 12,596 unique tokens - Example stemmed tokens for inspection (e.g., ‘fumbling’ -> ‘fumbl’)

Lemmatization (WordNetLemmatizer)

```
# Initializing the lemmatizer
lemmatizer = WordNetLemmatizer()

# Apply lemmatization to the tokenized text
lemmatized_texts = [[lemmatizer.lemmatize(word) for word in tokens] for tokens in tokenized_t
exts]

# Join tokens back into strings for compatibility with Keras Tokenizer.
lemmatized_texts_joined = [' '.join(tokens) for tokens in lemmatized_texts]

# Tokenizer settings.
MAX_WORDS = 10000
MAX_LEN = 150

tokenizer_lem = Tokenizer(num_words=MAX_WORDS, oov_token="<OOV>")
tokenizer_lem.fit_on_texts(lemmatized_texts_joined)

# Converting texts to sequences.
X_lem = tokenizer_lem.texts_to_sequences(lemmatized_texts_joined)

# Padded sequences.
X_lem = pad_sequences(X_lem, maxlen=MAX_LEN, padding='post', truncating='post')

# Labels
y = df_pandas['author_id'].values
```

Observed outputs (lemmatization): - Vocabulary size after lemmatization: 17,615 unique tokens

Vocabulary comparison and sequence-length distributions

```
original_vocab = set([word for tokens in tokenized_texts for word in tokens])
stemmed_vocab = set([word for tokens in stemmed_texts for word in tokens])
lemmatized_vocab = set([word for tokens in lemmatized_texts for word in tokens])

print("Original vocabulary size:", len(original_vocab))
print("Stemmed vocabulary size:", len(stemmed_vocab))
print("Lemmatized vocabulary size:", len(lemmatized_vocab))
```

Results: Original 21,124 | Stemmed 12,643 | Lemmatized 19,090

Sequence-length histograms were generated to show how preprocessing affects token counts per instance.

Word Clouds (EDA visuals)

The notebook generated word clouds for the original, stemmed, and lemmatized tokens to visualise the most frequent tokens for each preprocessing variant. These were used as qualitative checks that preprocessing behaved as intended.

Model architecture & training

This section shows the model and training function exactly as used in your experiments.

```
# Model builder (dense baseline used in experiments)
def build_model(input_dim: int):
    model = Sequential()
    model.add(Dense(128, activation="relu", input_dim=input_dim))
    model.add(Dropout(0.3))
    model.add(Dense(64, activation="relu"))
    model.add(Dense(3, activation="softmax"))

    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )
    return model

# Training function
def train_and_evaluate(X, y, label: str):
    print(f"
--- Training model with {label} tokens ---")
```

```
    # 1. Split
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    print("Train distribution:
", pd.Series(y_train).value_counts(normalize=True))
    print("Val distribution:
", pd.Series(y_val).value_counts(normalize=True))

    # 2. Class weights
    classes = np.unique(y)
    class_weights = compute_class_weight("balanced", classes=classes, y=y)
    class_weights_dict = dict(zip(classes, class_weights))
    print("Class Weights:", class_weights_dict)

    # 3. Build model
    model = build_model(X.shape[1])

    # 4. Train
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=10,
        batch_size=64,
        class_weight=class_weights_dict,
        verbose=1
    )

    # 5. Predictions
    y_pred_prob = model.predict(X_val)
    y_pred = y_pred_prob.argmax(axis=1)

    # 6. Evaluation
    acc = accuracy_score(y_val, y_pred)
    report = classification_report(y_val, y_pred, output_dict=True)

    print(f"
Classification Report for {label}:")
    print(classification_report(y_val, y_pred))

    return {
    "Preprocessing": label,
    "Accuracy": acc,
    "Precision": report["weighted avg"]["precision"],
    "Recall": report["weighted avg"]["recall"],
```

```
    "F1": report["weighted avg"]["f1-score"]
    }, (y_val, y_pred, y_pred_prob, history)
```

Models were trained for each preprocessing variant and outputs captured.

Results (as recorded in the notebook)

Below are the key metrics and classification reports captured for each preprocessing run. These are verbatim from the notebook outputs.

Original tokens — classification report (validation)

	precision	recall	f1-score	support
0	0.39	0.21	0.28	824
1	0.36	0.08	0.13	584
2	0.30	0.70	0.41	605

accuracy 0.32 (2013 samples)
weighted avg precision 0.35 recall 0.32 f1-score 0.28

Stemmed tokens — classification report (validation)

0	0.39	0.24	0.30	824
1	0.36	0.12	0.18	584
2	0.31	0.67	0.42	605

accuracy 0.33 (2013 samples)
weighted avg precision 0.36 recall 0.33 f1-score 0.30

Lemmatized tokens — classification report (validation)

0	0.40	0.21	0.28	824
1	0.33	0.19	0.24	584
2	0.30	0.61	0.40	605

accuracy 0.32 (2013 samples)
weighted avg precision 0.35 recall 0.32 f1-score 0.30

Aggregated results table (notebook results_df)

	Preprocessing	Accuracy	Precision	Recall	F1
0	lemmatized	0.326379	0.354246	0.326379	0.273387
1	original	0.405365	0.361766	0.405365	0.288427
2	stemmed	0.357178	0.372751	0.357178	0.313641
3	lemmatized	0.338798	0.379264	0.338798	0.311409
4	original	0.320914	0.352177	0.320914	0.276897
5	stemmed	0.334327	0.356057	0.334327	0.301122
6	lemmatized	0.324888	0.350613	0.324888	0.303638

Note: The results_df includes multiple rows due to multiple runs or aggregations in the notebook. For plotting summaries, averaged or chosen representative scores should be used. In the next section I produce visual comparisons using the provided numbers.

Visual comparisons (Accuracy and F1-score)

The notebook plotted Accuracy and weighted F1-score by preprocessing type. These visuals help compare which preprocessing produced better overall or balanced outcomes. The PDF export includes the same bar charts.

(Bar charts and training curve visuals are included in the downloadable PDF produced alongside this report.)

Discussion & Interpretation

Key observations: 1. **Original tokens preserve author-specific idiosyncrasies** (spelling, punctuation, rare word choice). This likely explains why the original token run achieves higher accuracy in some runs — authorship is often signalled by surface-level phenomena.

2. **Stemming reduces vocabulary size dramatically (~40%)**, which reduces sparsity and helps the model generalize across morphological variants. This often improved weighted F1 (balanced detection across classes), but sometimes reduces accuracy when author-specific word forms are important.

3. **Lemmatization offers a reasonable compromise** — it reduces redundancy while keeping tokens interpretable. It did not improve performance substantially in this experiment, suggesting author signals may lie outside normalized lemmas.

4. **Model capacity & architecture matters.** The baseline Dense network used here is simple. Training curves and plateauing losses suggest underfitting. For stylometry, stronger sequence-aware models (LSTMs/GRUs/Bi-LSTMs or transformer-based) and pretrained embeddings (GloVe/fastText) are likely to yield better results.

Practical implications: choose preprocessing based on the task: - For stylometry, **retain raw tokens** when surface forms matter. - Use **stemming** when vocabulary size is a critical problem and stylistic nuance is less important. - Use **lemmatization** when you want both interpretability and some reduction in sparsity.

Conclusion

Conclusion: Raw tokens produced the highest accuracy in some runs, indicating the importance of surface-features in authorship attribution. Stemming improved class-balance metrics (F1), while lemmatization produced intermediate results.

Recommended next steps: - Train a richer model (Embedding + Bidirectional LSTM or transformer) and compare. - Use pretrained embeddings (GloVe) to inject semantic similarity. - Add character-level features to capture punctuation and orthographic style. - Run stratified cross-validation and report mean ± std for metrics.

Appendix A — Full code (concise)

The notebook contains all the code blocks shown above and more (Spark ingestion, EDA plots, Zipf plots, etc.). For brevity they are not repeated here verbatim; the report shows the most critical sections.

References

christopher22 (2017) *Stylometry: Identify authors by sentence structure*, *Kaggle*. Available at: https://www.kaggle.com/code/christopher22/stylometry-identify-authors-by-sentence-structure (Accessed: 29 September 2025). – DATASET SOURCE

GeeksforGeeks (2025a) *Porter stemmer technique in natural language processing*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/nlp/porter-stemmer-technique-in-natural-language-processing/ (Accessed: 27 September 2025).

GeeksforGeeks (2025b) *Zipf's Law*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/nlp/zipfs-law/ (Accessed: 27 September 2025).

Laramée, F.D. (2018) *Introduction to stylometry with python*, *Programming Historian*. Available at: https://programminghistorian.org/en/lessons/introduction-to-stylometry-with-python (Accessed: 29 September 2025).