# Analysis Report

## Overview

Raw counts, TF-IDF, LSA, and Word2Vec are different methods for preparing text data for classification using Logistic Regression. The document classification task is to distinguish between "Sense and Sensibility" by Jane Austen and "Alice's Adventures in Wonderland" by Lewis Carroll.

## Performance

1. **Raw Counts**:
   - Training accuracy: 98.87%
   - Testing accuracy: 96.87%
2. **TF-IDF**:
   - Training accuracy: 99.93%
   - Testing accuracy: 97.16%
3. **LSA**:
   - Training accuracy: 96.77%
   - Testing accuracy: 95.07%
4. **Word2Vec**:
   - Training accuracy: 93.70%
   - Testing accuracy: 91.34%

## Comparison

1. **Raw Counts vs. TF-IDF**: The TF-IDF method slightly outperformed raw counts on the test data. This is expected because raw counts consider only the frequency of terms within a document without considering the context or the order of terms, while TF-IDF not only considers the frequency of a term in a document but also its uniqueness across all documents. This gives more weight to terms that are more distinguishing between the two authors, making the classification a bit more effective. However, it is worth noting that the training accuracy for TF-IDF is very high (almost 100%), which could suggest that the model might be overfitting to the training data.

2. **Raw Counts and TF-IDF vs. LSA**: LSA showed a decrease in performance compared to raw counts and TF-IDF. LSA is a technique for reducing dimensionality that may lead to the loss of significant information, especially in sparse datasets

where it might not perform optimally. While LSA can capture some semantic relationships between words, it may lose critical information for distinguishing between the two authors, leading to lower accuracy. Furthermore, the choice of 300 dimensions may not be precisely tuned for this particular dataset, potentially contributing to decreased accuracy.

3. **Raw Counts, TF-IDF, and LSA vs. Word2Vec**: Word2Vec had the lowest performance of all methods. This might be because Word2Vec represents words in a continuous vector space where semantically similar words are mapped to nearby points. While this is effective for capturing semantic similarities, it might not be as effective for authorship attribution where specific word choices and stylistic features are more important. Additionally, the pre-trained Word2Vec model is based on Google News data and uses 300-dimensional vectors. This generic training corpus may not include the domain-specific lexicon required for this particular classification task. Moreover, if the fixed 300-dimensional setting is not optimal for this dataset at hand, it can further diminish accuracy. Additionally, the predetermined context window size used by the Google News model to capture word co-occurrences could result in suboptimal representations for this specific task, potentially contributing to the reduced accuracy observed.

## Conclusion

The accuracy metric indicates how well the model correctly classifies the given sentences. Among the methods, TF-IDF yielded the best training and testing accuracy of 97.16%, suggesting it effectively captured distinguishing features between the two authors by emphasizing the importance of specific terms while diminishing the weight of frequently occurring but less informative terms. Raw counts provided the second-best training and testing accuracy, which works well as a direct and strong baseline when the frequency of word occurrence itself is a good indicator of authorship. LSA has the second lowest training and testing accuracy. Its dimensionality reduction can lead to the loss of vital information, particularly when dealing with sparse datasets. Word2Vec underperformed with the lowest test accuracy of 91.34%. This might be because Word2Vec focuses on semantic meanings, which may not be as crucial for authorship attribution as specific stylistic choices. Moreover, Word2Vec's training on Google News articles could miss nuances pertinent to this task. The fixed size of the context window used in the pre-trained model might also be misaligned with the data, causing suboptimal word or phrase representations. Furthermore, the chosen dimensionality of 300 for both LSA and Word2Vec may not be optimal for this dataset. It is notable that for all methods, the training accuracy was higher than the testing accuracy. This discrepancy hints that the models might be fitting closely to the training data, with TF-IDF showing an almost perfect training accuracy of 99.93%, raising concerns about potential overfitting.

In [ ]:
```python
"""Compare token/document vectors for classification."""
import random
from typing import List, Mapping, Optional, Sequence
import gensim
import nltk
import numpy as np
from numpy.typing import NDArray
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import LogisticRegression

FloatArray = NDArray[np.float64]
import gensim.downloader as api

# Load Google's pre-trained Word2Vec model.

model = api.load("word2vec-google-news-300")
# print(api.info())  # show info about available models/datasets

# Un-comment this to fix the random seed
random.seed(31)

austen = nltk.corpus.gutenberg.sents("austen-sense.txt")
carroll = nltk.corpus.gutenberg.sents("carroll-alice.txt")
vocabulary = sorted(
    set(token for sentence in austen + carroll for token in sentence)
) + [None]

vocabulary_map = {token: idx for idx, token in enumerate(vocabulary)}


def onehot(
    vocabulary_map: Mapping[Optional[str], int], token: Optional[str]
) -> FloatArray:
    """Generate the one-hot encoding for the provided token in the provided
    embedding = np.zeros((len(vocabulary_map),))
    idx = vocabulary_map.get(token, len(vocabulary_map) - 1)
    embedding[idx] = 1
    return embedding


def sum_token_embeddings(
    token_embeddings: Sequence[FloatArray],
) -> FloatArray:
    """Sum the token embeddings."""
    total: FloatArray = np.array(token_embeddings).sum(axis=0)
    return total


def split_train_test(
    X: FloatArray, y: FloatArray, test_percent: float = 10
) -> tuple[FloatArray, FloatArray, FloatArray, FloatArray]:
    """Split data into training and testing sets."""
    N = len(y)
    data_idx = list(range(N))
    random.shuffle(data_idx)
    break_idx = round(test_percent / 100 * N)
    training_idx = data_idx[break_idx:]
    testing_idx = data_idx[:break_idx]
    X_train = X[training_idx, :]
    y_train = y[training_idx]
    X_test = X[testing_idx, :]
    y_test = y[testing_idx]
    return X_train, y_train, X_test, y_test


def generate_data_token_counts(
    h0_documents: list[list[str]], h1_documents: list[list[str]]
) -> tuple[FloatArray, FloatArray, FloatArray, FloatArray]:
    """Generate training and testing data with raw token counts."""
    X: FloatArray = np.array(
        [
            sum_token_embeddings([onehot(vocabulary_map, token) for token in
            for sentence in h0_documents
        ]
        + [
            sum_token_embeddings([onehot(vocabulary_map, token) for token in
            for sentence in h1_documents
        ]
    )
    y: FloatArray = np.array(
        [0 for sentence in h0_documents] + [1 for sentence in h1_documents]
    )
    return split_train_test(X, y)


def generate_data_tfidf(
    h0_documents: list[list[str]], h1_documents: list[list[str]]
) -> tuple[FloatArray, FloatArray, FloatArray, FloatArray]:
    """Generate training and testing data with TF-IDF scaling."""
    X_train, y_train, X_test, y_test = generate_data_token_counts(
        h0_documents, h1_documents
    )
    tfidf = TfidfTransformer(norm=None).fit(X_train)
    X_train = tfidf.transform(X_train)
    X_test = tfidf.transform(X_test)
    return X_train, y_train, X_test, y_test


def generate_data_lsa(
    h0_documents: list[list[str]], h1_documents: list[list[str]]
) -> tuple[FloatArray, FloatArray, FloatArray, FloatArray]:
    """Generate training and testing data with LSA."""
    X_train, y_train, X_test, y_test = generate_data_token_counts(
        h0_documents, h1_documents
    )
    lsa = TruncatedSVD(n_components=300).fit(X_train)
    X_train = lsa.transform(X_train)
    X_test = lsa.transform(X_test)
    return X_train, y_train, X_test, y_test
```

```python
def generate_data_word2vec(
    h0_documents: list[list[str]], h1_documents: list[list[str]]
) -> tuple[FloatArray, FloatArray, FloatArray, FloatArray]:
    """Generate training and testing data with word2vec."""
    # Load pretrained word2vec model from gensim
    model = api.load("word2vec-google-news-300")

    def get_document_vector(sentence: list[str]) -> NDArray:
        """Return document vector by summing word vectors."""
        vectors = [model[word] for word in sentence if word in model.key_to_
        if vectors:
            return np.sum(vectors, axis=0)
        else:
            return np.zeros(
                300
            )  # return zero vector if no word in the document has a pretrai

    # Produce document vectors for each sentence
    X = np.array(
        [get_document_vector(sentence) for sentence in h0_documents + h1_doc
    )
    y = np.array([0 for sentence in h0_documents] + [1 for sentence in h1_do
    return split_train_test(X, y)


def run_experiment() -> None:
    """Compare performance with different embeddiings."""
    X_train, y_train, X_test, y_test = generate_data_token_counts(austen, ca
    clf = LogisticRegression(random_state=0, max_iter=1000).fit(X_train, y_t
    print("raw counts (train):", clf.score(X_train, y_train))
    print("raw_counts (test):", clf.score(X_test, y_test))
    X_train, y_train, X_test, y_test = generate_data_tfidf(austen, carroll)
    clf = LogisticRegression(random_state=0, max_iter=1000).fit(X_train, y_t
    print("tfidf (train):", clf.score(X_train, y_train))
    print("tfidf (test):", clf.score(X_test, y_test))
    X_train, y_train, X_test, y_test = generate_data_lsa(austen, carroll)
    clf = LogisticRegression(random_state=0, max_iter=1000).fit(X_train, y_t
    print("lsa (train):", clf.score(X_train, y_train))
    print("lsa (test):", clf.score(X_test, y_test))
    X_train, y_train, X_test, y_test = generate_data_word2vec(austen, carrol
    clf = LogisticRegression(random_state=0, max_iter=1000).fit(X_train, y_t
    print("word2vec (train):", clf.score(X_train, y_train))
    print("word2vec (test):", clf.score(X_test, y_test))


if __name__ == "__main__":
    run_experiment()
```

```
raw counts (train): 0.9887267904509284
raw_counts (test): 0.9686567164179104
tfidf (train): 0.9993368700265252
tfidf (test): 0.9716417910447761
lsa (train): 0.9676724137931034
lsa (test): 0.9507462686567164
word2vec (train): 0.9370026525198939
word2vec (test): 0.9134328358208955
```