International Institute of Information Technology Bangalore

AIM 829 - Natural Language Processing, 2025

Final Project Report

Creating an Embedding Scheme for Financial Terms

Course Instructor:
Professor Tulika Saha
IIITB

Submitted By :
Prabhav Pandey (MT2024115)

Contents

1 Introduction

This project focuses on creating specialized word embeddings for financial terms by training Word2Vec and FastText models on financial sentiment data. The goal is to develop embeddings that better capture financial semantics compared to general-purpose embeddings. We evaluate these embeddings using three classification models: Logistic Regression, Random Forest, and CNN.

2 Dataset Preparation

2.1 Data Sources

We combined two financial sentiment datasets:

- **Financial Sentiment Analysis Dataset** from Kaggle [1]
  - Shape: 5,842 samples (original)
  - Columns: `Sentence` (financial text), `Sentiment` (positive/negative/neutral)
  - Example: ”The GeoSolutions technology will leverage...”, ”positive”
  - Contains financial news headlines and sentences
- **Financial Tweets Sentiment Dataset** from Hugging Face [2]
  - Shape: 38,091 samples (original)
  - Columns: `tweet` (financial tweet text), `sentiment` (0:negative, 1:neutral, 2:positive), `url`
  - Example: ”$BYND - JPMorgan reels in expectations...”, 2 (positive)
  - Contains Twitter messages about stocks and companies

After merging and preprocessing, our combined dataset contained 43,933 samples with unified sentiment labels (-1:negative, 0:neutral, 1:positive).

2.2 Preprocessing

Key preprocessing steps included:

- URL removal
- Handling ticker symbols (e.g., $AAPL → aapl) handling (e.g., 25% → 25percent)
- Lemmatization and stopword removal
- Financial-specific cleaning (retaining terms like ”percent”)

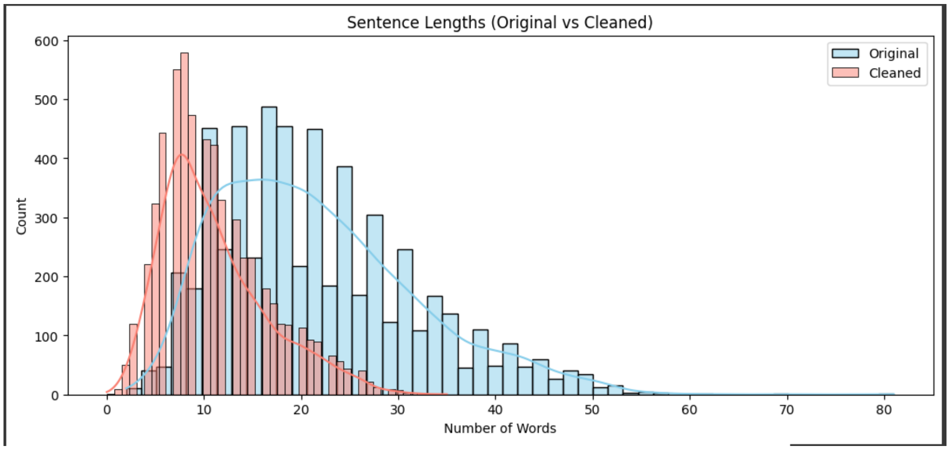

Figure 1: Sentence length distribution before and after preprocessing

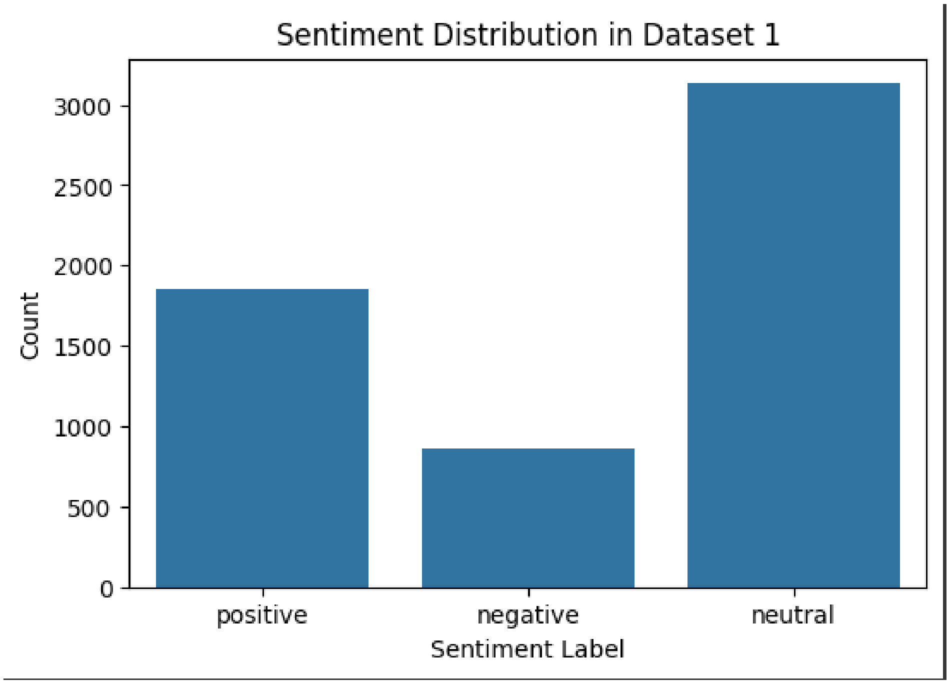

Figure 2: Sentiment distribution in the merged dataset

3 Vocabulary Construction

To enable training word embeddings, we constructed a vocabulary from the merged financial sentiment dataset. The process involved the following steps:

- The `Cleaned` column, containing preprocessed text, was tokenized using NLTK's `word_tokenize` function.
- A frequency count of all tokens was computed using Python's `Counter` to capture word usage across the dataset.
- Special tokens `<PAD>` (for sequence padding) and `<UNK>` (for unknown words) were prepended to the vocabulary.
- Two mappings were constructed: `word2idx`, mapping words to unique indices, and `idx2word`, the reverse mapping.
- The final vocabulary dictionary, containing a total of 48,945 terms, was serialized and saved in JSON format as `vocabulary.json`.

4 Embedding Training

4.1 Word2Vec Implementation

To capture word semantics, we employed the Skip-Gram variant of Word2Vec. Our training process was structured as follows:

- **Preprocessing and Pair Generation:** We constructed center-context word pairs using a sliding window of size 2. This resulted in 1,132,522 training pairs.
- **Model Architecture:** Each center word was mapped to a dense 100-dimensional embedding. These embeddings were used to predict context words over the full vocabulary.
- **Training Setup:** We trained the model over 5 epochs, using negative sampling and cross-entropy loss for optimization.
- **Objective:** The model learned embeddings such that words appearing in similar contexts had similar vector representations.

4.2 FastText Implementation

We further enhanced the embedding quality using FastText, which extends Word2Vec by incorporating subword information:

- **Subword Generation:** Words were broken down into character n-grams, with lengths ranging from 3 to 6 characters.
- **Vocabulary Augmentation:** This produced 48,945 unique n-gram units, which were used to construct each word's representation by averaging its n-gram embeddings.
- **Architecture Reuse:** The same network architecture and training process as Word2Vec was used, except that input embeddings were aggregated from the subword units.

This approach helped in capturing morphological patterns and significantly improved representation for rare and out-of-vocabulary words.

4.3 Visualization

To assess the quality of the learned embeddings, we used t-SNE dimensionality reduction. Figure 4 presents a 2D visualization of both Word2Vec and FastText embeddings. Notably, FastText produced tighter clusters, indicating improved semantic grouping due to its subword-level understanding.

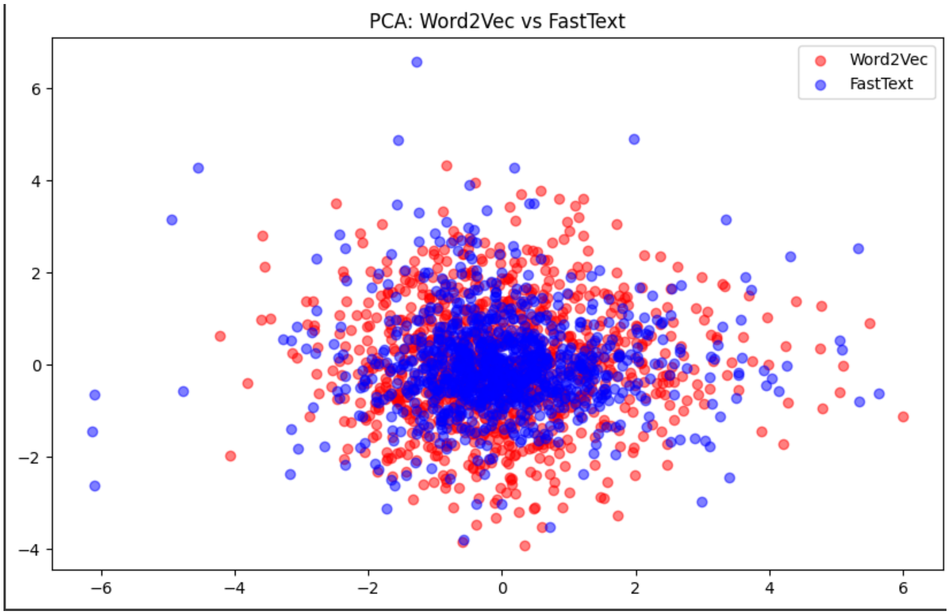

Figure 3: t-SNE visualization of Word2Vec and FastText embeddings

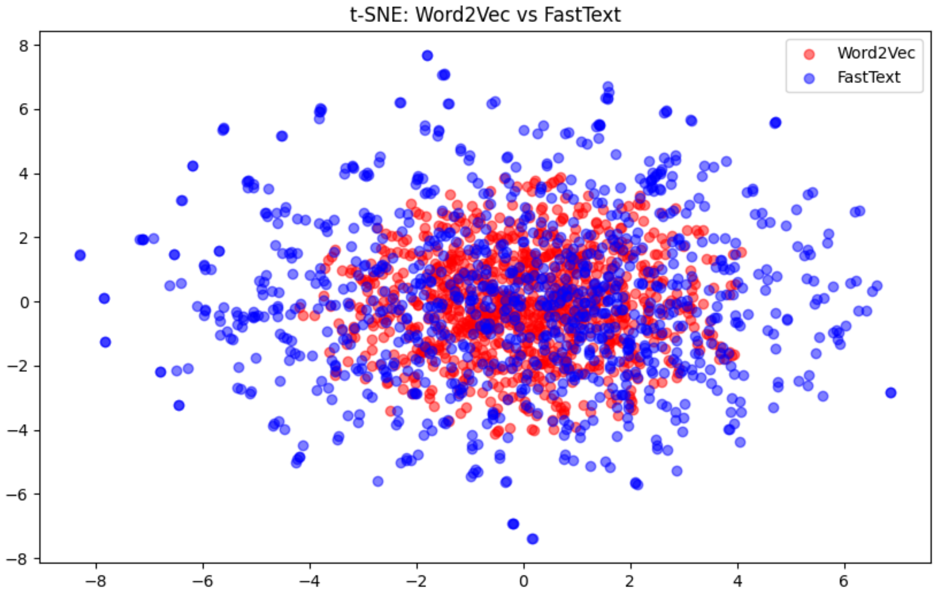

Figure 4: PCA visualization of Word2Vec and FastText embeddings

5 Model Implementation

The classification pipeline involved the following steps:

1. **Data Preprocessing:**
   - Removed punctuation, numbers, and stopwords.
   - Converted all text to lowercase.
   - Tokenized each sentence using NLTK tokenizer.
   - Applied padding for models requiring fixed-length inputs (CNN).
2. **Embedding Generation:**
   - Used pre-trained Word2Vec and FastText embeddings.
   - For logistic regression and random forest, we computed sentence-level features by averaging word embeddings.
   - For CNN, full sequences of word embeddings were retained and padded to fixed length.
3. **Model Training and Evaluation:**
   - Dataset split into training and test sets (80/20 ratio).
   - Accuracy was the primary evaluation metric.
   - The following models were trained:

5.1 Logistic Regression

- Input: averaged word embeddings.
- Implementation: `sklearn.linear_model.LogisticRegression`.
- Results:
  - FastText accuracy: 56.67%
  - Word2Vec accuracy: 56.05%

5.2 Random Forest

- Input: averaged word embeddings.
- Implementation: `sklearn.ensemble.RandomForestClassifier` with 100 estimators.
- Results:
  - FastText accuracy: 60.91%
  - Word2Vec accuracy: 58.61%

5.3 Convolutional Neural Network (CNN)

- Input: padded word embeddings sequences.
- Architecture:
  - Embedding dimension as input channels.
  - Convolutional layers with kernel sizes 3, 4, 5.
  - 100 filters per kernel size.
  - Max-pooling applied to each filter output.
  - Dropout rate of 0.5 for regularization.
  - Fully connected layer for final classification.
- Results:
  - FastText accuracy: 62.46%
  - Word2Vec accuracy: 62.43%

```
class CNNModel(nn.Module):
    def __init__(self, input_dim, num_classes, num_filters=100,
                 kernel_sizes=[3, 4, 5]):
        super(CNNModel, self).__init__()
        self.convs = nn.ModuleList([
            nn.Conv1d(input_dim, num_filters, k) for k in
   kernel_sizes
        ])
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(len(kernel_sizes)*num_filters,
   num_classes)
```

Listing 1: CNN Model Definition

6 Results and Analysis

The results from our experiments demonstrate how different models perform when combined with Word2Vec and FastText embeddings. The key findings are discussed below:

- **CNN Performance:** The CNN model achieved the highest accuracy across both embedding types (62.46% for FastText and 62.43% for Word2Vec). This highlights CNN's ability to effectively capture local contextual features such as n-gram patterns via convolutional filters.
- **FastText vs. Word2Vec:** FastText embeddings consistently outperformed Word2Vec in all three models. This can be attributed to FastText's ability to incorporate subword information, making it more robust to rare and out-of-vocabulary words.
- **Classical Models:** Among the classical models, Random Forest surpassed Logistic Regression by a notable margin. The non-linear nature of Random Forest allows it to capture complex relationships in the embedding space, whereas Logistic Regression remains limited to linear decision boundaries.
- **Embedding Aggregation Effects:** For classical models, averaged word embeddings were used, which may lead to loss of sequential information. This could explain the improved performance of CNN, which processes the full sequence and preserves word order via convolutional layers.
- **Computational Considerations:** Although CNN yields higher accuracy, it requires significantly more computational resources and training time compared to Logistic Regression and Random Forest, which are more lightweight and easier to deploy.

Table 1: Model Performance Comparison (Accuracy %)

Model	**FastText**	**Word2Vec**
Logistic Regression	56.67	56.05
Random Forest	60.91	58.61
CNN	**62.46**	62.43

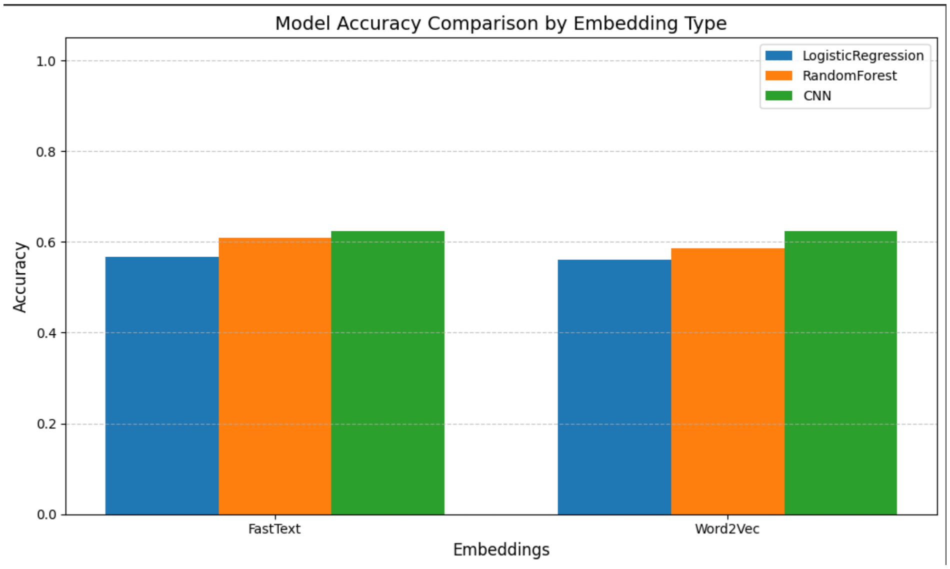

Figure 5: Comparison of model accuracies using FastText and Word2Vec embeddings

7 Conclusion

This project successfully:

- Created financial-specific Word2Vec and FastText embeddings
- Demonstrated CNN's superiority for text classification with embeddings
- Showed FastText's slight advantage over Word2Vec for financial text

Future work could explore:

- Larger financial corpora
- Transformer-based embeddings
- Domain-specific fine-tuning

References

[1] Bhatti, S. (2022). Financial Sentiment Analysis. `https://www.kaggle.com/datasets/sbhatti/financial-sentiment-analysis`

[2] Koornstra, T. (2021). Financial Tweets Sentiment. `https://huggingface.co/datasets/TimKoornstra/financial-tweets-sentiment`