Naïve-Bayes text classification model

Python Code Analysis Report

Victor Mai

April 4, 2025

**I acknowledge the use of Deepseek to refine word usage and polish grammar to fulfill the academic English requirements.*

1 Project Overview

This project implements a *naïve-bayes* text classification model and evaluates the performance of the implemented model in the Reuters dataset.

The project consists of two main parts:

1. **Train a classification model**: Apply Bayes rule to determine the class distribution possibility of the documents.
2. **Test and calculate the F1 score associated with**: Classify pre-processed test data using a trained classification model and measure model performance.

2 Train a classification model

2.1 Design Approach

- Uses `preprocess()` to preprocess the train.json and test.json with following steps:
    1. Replace `"&"`, `"<"`, `">"`, `"""`, and `"'"` with spaces. (They are HTML signals!)
    2. Replace all symbols `?`, `!`, `<`, `>`, `:`, `;`, `\n`, and `"` with spaces.
    3. Replace all apostrophes (`'`) with spaces, except for the pattern `s'` (possessive form).
    4. Convert all letters to lowercase.
    5. Apply `NLTK.word_tokenize()` for tokenization.
    6. For each token, remove redundant `^m` substrings at the end.
    7. For each token, remove redundant dots (`.`, `..`, `...`, etc.) at the end.
    8. Remove all tokens that do not contain at least one letter (a-z) or digit (0-9) or `$`.
    9. Apply `nltk.PorterStemmer` to the remaining words for stemming.
- Count the word frequency of train data and restore as **word_count.txt**.
- Select the first n words of the frequency (n is variable and will be used for dynamic testing) and save as **word_dict.txt**.
- Calculate the prior probability of each class and the posterior probability of each word in **word_dict.txt** (including `<UNKNOWN>` words, using Laplace smoothing).

2.2 Key Code of str parsing

```python
def preprocess_str(text):
    html_entities = {
        "&amp;": "&",
        "&lt;": "<",
        "&gt;": ">",
        "&quot;": "\"",
        "&apos;": "'"
    }
    for entity, replacement in html_entities.items():
        text = text.replace(entity, replacement)

    # Replace them with space
    symbols = r'[?!<>:;\n"]'
    text = re.sub(symbols, " ", text)

    # Replace single quotes with spaces, except "s'"
    text = re.sub(r"(?<!\w)'|'(?!\w|s')", " ", text)

    # Lowercase
    text = text.lower()

    # Tokenize
    tokens = word_tokenize(text)

    # Remove "^M" signal
    tokens = [re.sub(r'\^m$', '', token) for token in tokens]

    # Remove redundant "."s
    tokens = [re.sub(r'\.{2,}$', '', token) for token in tokens]
    tokens = [re.sub(r'\.$', '', token) for token in tokens]

    # Remove words that do not contain letters or numbers
    tokens = [token for token in tokens if re.search(r'[a-z0-9$]', token)]

    # PorterStemmer
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(token) for token in tokens]

    return tokens
```

Listing 1: Core of preprocessing

2.3 Pre-processing the JSON files

```python
def preprocess(inputfile, outputfile):
    with open(inputfile, 'r', encoding='utf-8') as f:
        inputdata = json.load(f)
        f.close()

    for each in tqdm(inputdata):
        each[2] = preprocess_str(each[2])

    with codecs.open(outputfile, 'w', encoding='utf-8') as fo:
        fo.write(json.dumps(inputdata))
        fo.close()

    return
```

Listing 2: Preprocess the json and save the words Informationlabel

2.4 Counting the frequency

```python
def count_word(inputfile, outputfile):
    with open(inputfile, 'r', encoding='utf-8') as f:
        inputdata = json.load(f)
        f.close()

    to_idx = {'crude': 0, 'grain': 1, 'money-fx': 2, 'acq': 3, 'earn': 4}
    counter = [0] * len(to_idx)

    mydict = {}
    for item in tqdm(inputdata):
        if len(item) < 3:
            continue

        category = item[1]
        words = item[2]

        try:
            idx = to_idx[category]
        except KeyError:
            continue
        counter[idx] += 1

        for word in words:
            if word not in mydict:
                mydict[word] = [0] * len(to_idx)
            mydict[word][idx] += 1

    with open(outputfile, 'w', encoding='utf-8') as fo:
        line = '{} {} {} {} {}\n'.format(*counter)
        fo.write(line)
        for key, item in tqdm(mydict.items()):
            line = '{} {} {} {} {} {}\n'.format(key, *item)
            fo.write(line)
        fo.close()
    return
```

Listing 3: Count the frequency of words

2.5 Selecting the features

```python
def feature_selection(inputfile, threshold, outputfile):
    word_count = {}
    with open(inputfile, 'r', encoding='utf-8') as f:
        for line in tqdm(f):
            line = line.strip()
            words = line.split(' ')
            if len(words) < 6:
                continue  # First line
            word_count[words[0]] = sum(int(x) for x in words[1:])
        f.close()

    top_keys = heapq.nlargest(min(len(word_count), threshold), word_count.keys(), key=word_count.get)
    top_keys.sort()

    with open(outputfile, 'w', encoding='utf-8') as fo:
        for key in top_keys:
            fo.write('{}\n'.format(key))
        fo.close()
    return
```

Listing 4: Select the features with threshold

2.6 Calculating probabilities

```
def calculate_probability(word_count, word_dict, outputfile):
    posterior_probability = {}

    with open(word_dict, 'r', encoding='utf-8') as fwd:
        for word in tqdm(fwd):
            word = word.strip()
            posterior_probability[word] = [Fraction(0, 1)] * 5
        fwd.close()

    prior_probability = [Fraction(0)] * 5
    class_word_count = [Fraction(0)] * 5
    one = Fraction(1, 1)
    v = Fraction(len(posterior_probability), 1)

    with open(word_count, 'r', encoding='utf-8') as fwc:
        for line in tqdm(fwc):
            line = line.strip()
            words = line.split(' ')
            if len(words) < 6:  # First line
                for i in range(len(words)):
                    prior_probability[i] = Fraction(int(words[i]))
                n_doc = sum(prior_probability, Fraction(0, 1))
                for i in range(5):
                    prior_probability[i] /= n_doc
            elif words[0] in posterior_probability:
                for i in range(1, len(words)):
                    posterior_probability[words[0]][i - 1] = Fraction(int(words
[i]), 1)
                    class_word_count[i - 1] += Fraction(int(words[i]), 1)
            else:
                continue
        fwc.close()

    for key in posterior_probability:
        for i in range(5):
            posterior_probability[key][i] += one
            posterior_probability[key][i] /= (class_word_count[i] + v)
    posterior_probability['<UNKNOWN>'] = [Fraction(1, class_word_count[i] + v +
 1)] * 5

    with open(outputfile, 'w', encoding='utf-8') as fo:
        fo.write(' '.join(str(pc) for pc in prior_probability) + '\n')
        for key, item in tqdm(posterior_probability.items()):
            fo.write(key + ' ' + ' '.join(str(post) for post in item) + '\n')
        fo.close()
    return
```

Listing 5: Calculate prior and posterior probabilities(including unknown words by using add-one)

3 Test and calculate the F1 score associated with

3.1 Design Approach

Text processing involves the following 2 steps:

1. Use `classify()` to classify test cases and save the results as `.txt` files.

2. Use f1_score() to calculate the F1 score of the trained model.

3.2 Classification Method

```
def classify(probability, testset, outputfile):
    #   Output the result to the output file in the format required
    classes = ['crude', 'grain', 'money-fx', 'acq', 'earn']
    classes_prob = np.zeros(5)  #     np    . a r r a y  [0.0]*5

    prob = {}
    with open(probability, 'r', encoding='utf-8') as f:
        for line in tqdm(f):
            line = line.strip()
            items = line.split(' ')

            if len(items) < 6:
                for i in range(len(items)):
                    frac = Fraction(items[i])
                    classes_prob[i] = math.log(frac.numerator) - math.log(frac.denominator)
                continue

            prob[items[0]] = np.array([math.log(Fraction(item).numerator) - math.log(Fraction(item).denominator)
                                        for item in items[1:6]])

    results = []
    with open(testset, 'r', encoding='utf-8') as f:
        testdata = json.load(f)
        for text in tqdm(testdata):
            final_prob = classes_prob.copy()

            for word in text[2]:
                if word in prob:
                    final_prob += prob[word]
                else:
                    final_prob += prob['<UNKNOWN>']

            predicted_class = classes[final_prob.argmax()]
            results.append((text[0], predicted_class))
        f.close()

    with open(outputfile, 'w', encoding='utf-8') as fo:
        for result in results:
            fo.write('{} {}\n'.format(result[0], result[1]))

    return
```

Listing 6: Classify the testset

3.3 Calculate the F1 score

```
def f1_score(testset, classification_result, average='micro'):
    y_true, y_pred = [], []
    with open(testset, 'r', encoding='utf-8') as f:
        testdata = json.load(f)
        for text in tqdm(testdata):
            y_true.append(text[1])
        f.close()

    with open(classification_result, 'r', encoding='utf-8') as f:
        for result in tqdm(f):
```

```
            result = result.strip()
            case, predicted_class = result.split(' ')
            y_pred.append(predicted_class)

    micro_average_f1 = sklearn_f1_score(np.array(y_true), np.array(y_pred),
    average=average)
    return micro_average_f1
```

Listing 7: Calculate the F1 score

4 Result

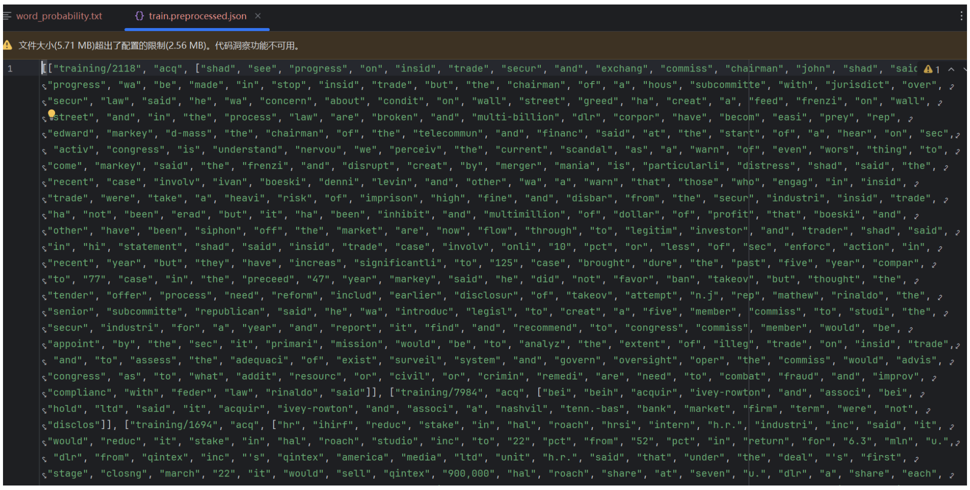
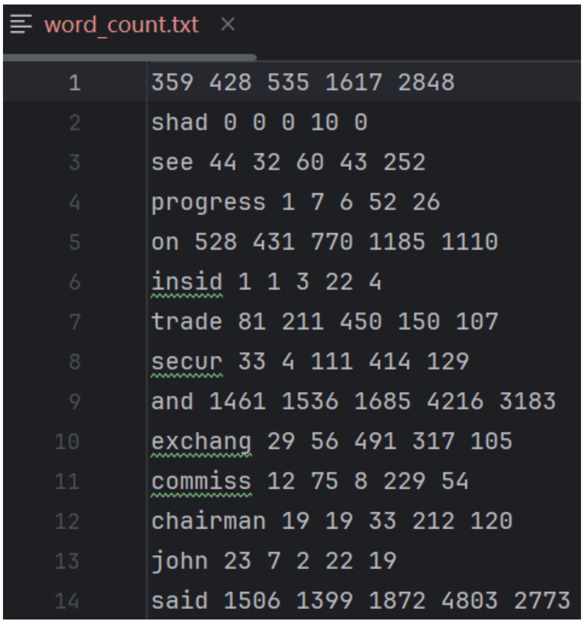


Figure 1: train.preprocessed.json.png

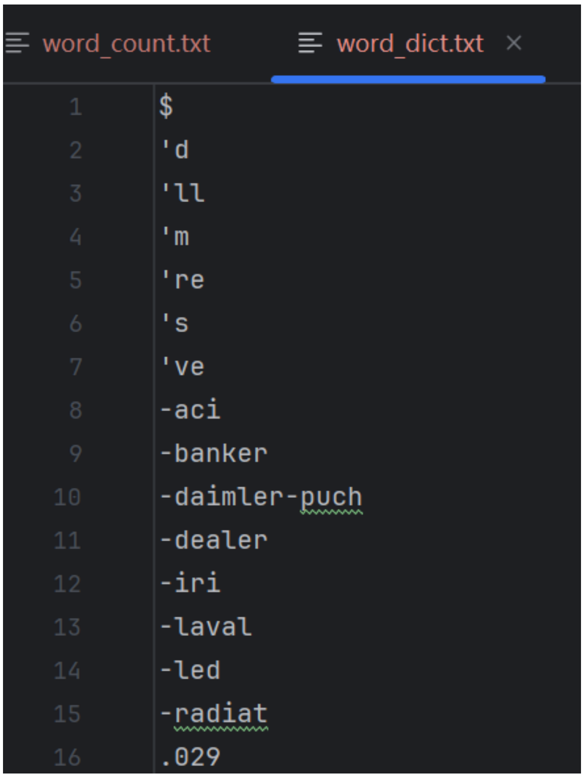
Figure 2: test.preprocessed.json.png

5 Addition works

If the pre-loaded dictionaries are too large, it will consume too much performance. In order to better study the relationship between the model's performance and the number of features, I customized a method to compute the F1 of the model trained with different numbers of features, and plotted the

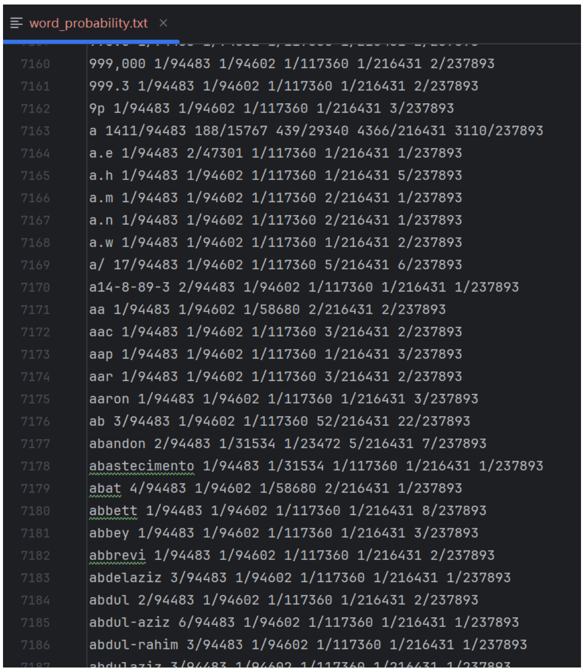


(a) word_count.png

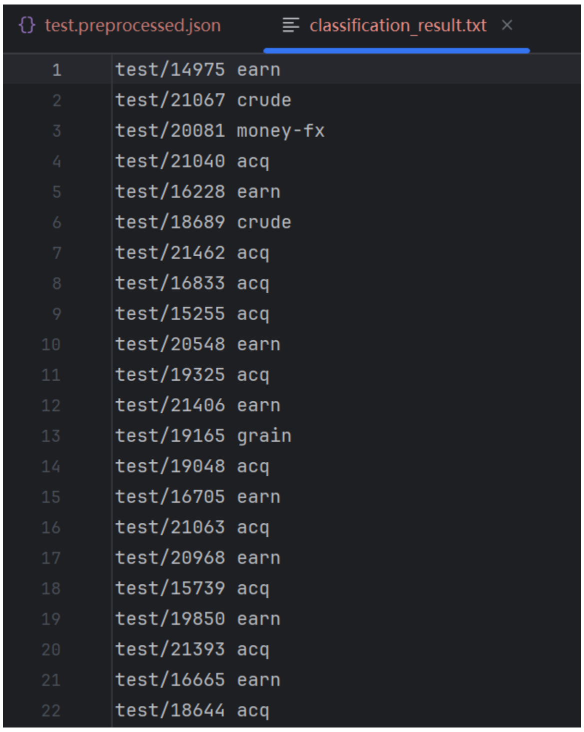


(b) word_dict.png

Figure 3: word_count and word_dict



(a) word_probability.png



(b) classification_result.png

Figure 4: word_probability and classification_result

corresponding curve graph 5 (log scale).

∗ The number of features ranges from 10 to 20000 (log scale).

5.1 Relationship between the model's performance and the number of features

```python
def plt_performance(start_n: int = 10, end_n: int = 20000, point_n: int = 50):
    preprocess('train.json', 'train.preprocessed.json')
    preprocess('test.json', 'test.preprocessed.json')
    count_word('train.preprocessed.json', 'word_count.txt')
    num_feature = np.logspace(
        np.log10(start_n),
        np.log10(end_n),
        num=point_n,
        dtype=int
    ).tolist()
    scores = []

    for n in num_feature:
        feature_selection('word_count.txt', n, 'word_dict.txt')
        calculate_probability('word_count.txt', 'word_dict.txt', 'word_probability.txt')
        classify('word_probability.txt', 'test.preprocessed.json', 'classification_result.txt')
        score = f1_score('test.json', 'classification_result.txt')
        scores.append(score)
        print('{} features, F1 score: {:.4f}'.format(n, score))

    plt.figure(figsize=(12, 6))
    plt.plot(num_feature, scores, 'bo-', linewidth=2, markersize=8)
    plt.xscale('log')
    plt.xlabel('Number of features (log scale)')
    plt.ylabel('F1 Score')
    plt.title('Model Performance')
    plt.grid(True, linestyle='--', alpha=0.7)

    max_score = max(scores)
    max_idx = scores.index(max_score)
    plt.annotate(f'Max F1: {max_score:.4f}\n(n={num_feature[max_idx]})',
                 xy=(num_feature[max_idx], max_score),
                 xytext=(10, 10), textcoords='offset points',
                 bbox=dict(boxstyle='round,pad=0.5', fc='yellow', alpha=0.5),
                 arrowprops=dict(arrowstyle='->'))

    plt.tight_layout()
    plt.savefig('feature_vs_f1.png', dpi=300)
    plt.show()
```

Listing 8: plt_performance.py

6 Conclusion

- Through this project, I clearly realised the importance of feature selection in text classification. Also, get a feel for the simplicity and practicality of Bayes RULE.
- In terms of model evaluation, the adoption of the F1 score as a core metric is of practical significance as it takes into account both Precision and Recall, and is particularly suitable for dealing with datasets with unbalanced categories.

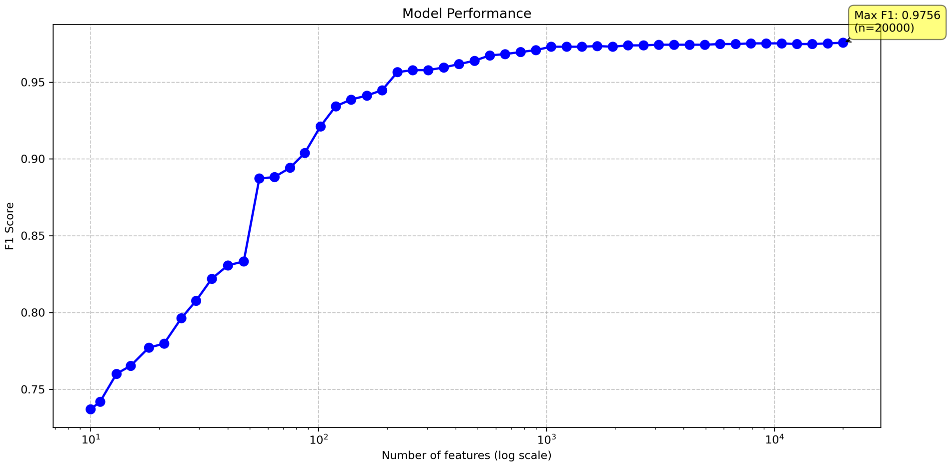


Figure 5: feature_vs_f1.png

- The highest F1 score obtained from the experiment is 0.9756, which occurs when the number of features is about 20000. (shown as Figure 5) However, if we extract only 200 feature with highest frequency, F1 can also reach the level of 0.95.
- Through this project, I have not only mastered the principle of implementing the plain Bayesian classifier, but also deeply understood the fundamental role of text preprocessing for NLP tasks. Meanwhile, I systematically practiced the complete process of machine learning project: from data preparation, feature engineering, model implementation to performance evaluation, which is a good exercise for my engineering ability.