Sentiment Analysis on News Headlines

A Comprehensive Study of Different Machine Learning Models

Author: Team Shiba Inu
Date: March 18, 2025

Project Structure and GitHub Repository Organization

GitHub Repository Link: ml-course-shibainu

```
kchan139-ml-course-shibainu/
├── README.md
├── LICENSE
├── requirements.txt
├── dataset/                    # Data storage
├── models/                     # Model artifacts
├── notebooks/                  # Jupyter notebooks
├── reports/
├── src/                        # Source code
│   ├── __init__.py
│   ├── config.py
│   ├── data/
│   │   ├── __init__.py
│   │   ├── make_dataset.py
│   │   └── preprocess.py
│   ├── features/
│   │   ├── __init__.py
│   │   └── build_features.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── predict_model.py
│   │   └── train_model.py
│   └── visualization/
│       ├── __init__.py
│       └── visualize.py
└── tests/                     # Unit tests
    ├── __init__.py
    ├── test_data.py
    └── test_models.py
```

Branching Strategy

Our repository implements a modified Git Flow workflow:

- **main**: Production-ready code, protected branch
- **develop**: Integration branch for feature development
- **feature/***: Individual feature branches (e.g., feature/naive-bayes)

Team Members and Workload Distribution

Member	ID	Responsibilities	Assigned Model
Huynh Kiet Khai	2252338	Model optimization	Decision Tree
Tran Dinh Dang Khoa	2211649	Repository setup, project coordination	Neural Network
Ha Tuan Khang	2252289	Validation strategy	Naive Bayes
Phan Chi Vy	2252938	Data preprocessing, feature engineering	Bayesian Network
Nguyen Duc Tam	2252734	Data Visualization	Hidden Markov Model

Each team member's workflow follows these stages:

1. Research: Understanding algorithms and relevant implementations
2. Implementation: Coding the assigned model in src/models/
3. Training: Creating notebooks for model development
4. Evaluation: Testing performance on validation data
5. Documentation: Adding comments and updating documentation
6. Integration: Merging with the main codebase via Pull Request

Model Development Cycle

Each of the five models (Decision Trees, Neural Networks, Naive Bayes, Bayesian Networks, and Hidden Markov Models) follows a standardized development process:

1. Initial implementation in isolated feature branches
2. Regular code reviews by at least one other team member
3. Documentation including theoretical background and practical usage
4. Testing before merging to develop branch

Introduction

This report documents Phase 1 of our machine learning project, focusing on the implementation and comparative analysis of various ML models for sentiment analysis of news headlines. Our goal is to demonstrate a practical understanding of ML models covered in Chapters 2-6 of the course curriculum. We've implemented Decision Trees, Neural Networks, Naive Bayes, and Graphical Models (Bayesian Networks and Hidden Markov Models) to classify news headlines into positive, neutral, or negative sentiment categories.

Problem Statement & Dataset

Problem Statement

We aim to create a sentiment analysis system capable of accurately classifying news headlines into three sentiment categories: negative, neutral, and positive. This task is challenging due to the brevity of headlines, implied sentiment, and contextual nuances.

Dataset

Our dataset consists of news headlines labeled with sentiment scores. The preprocessing pipeline handles cleaning, tokenization, and feature extraction to prepare the data for various modeling approaches.

Challenges

- Short text length with limited contextual information
- Implicit sentiment not directly expressed through sentiment words
- Class imbalance with neutral headlines comprising the majority
- Need for different feature representations depending on the model type

Methodology

Data Preprocessing

- Text cleaning (lowercase conversion, punctuation removal)
- Stopword removal and lemmatization
- Feature extraction using vectorization techniques:
    - Count-based vectorization for Decision Trees and Naive Bayes
    - Word embeddings for Neural Networks
    - Binary feature presence for Bayesian Networks
    - Discretized features for specific probabilistic models

Model Selection

For Phase 1, we implemented:

1. Decision Trees: Feature-based classification with pruning
2. Neural Networks: RNN architecture with LSTM units for sequential data
3. Naive Bayes: Multinomial classifier with TF-IDF features
4. Bayesian Networks with structure learning
5. Hidden Markov Models for sequence data

Model Implementations

Decision Tree Model

```python
def train_decision_tree(self, X_train, y_train):
    # Define a parameter grid for hyperparameter tuning
    param_grid = {
        'max_depth': [None, 5, 10, 15, 20, 25],
        'min_samples_leaf': [1, 2, 4, 8, 16],
        'criterion': ['entropy', 'gini']
    }

    dt = DecisionTreeClassifier(random_state=42)
    grid_search = GridSearchCV(dt, param_grid, cv=5, scoring='accuracy')
    grid_search.fit(X_train, y_train)

    self.decision_tree_model = grid_search.best_estimator_

    # Save the trained model to the relative directory.
    save_path = os.path.join(MODEL_DIR, 'decision_tree_model.pkl')
    with open(save_path, 'wb') as f:
        pickle.dump(self.decision_tree_model, f)

    return self.decision_tree_model
```

Key Features:

- Grid search for hyperparameter optimization
- Pruning through max depth and min samples leaf parameters
- Model persistence for reusability
- Handles high-dimensional text features after vectorization

Neural Network Model

We implemented a bidirectional LSTM model specifically designed to handle class imbalance in sentiment data:

```
def train_neural_network(self, preprocessor=None, file_path=None,
max_words=3000,
                            embedding_dim=32, max_len=30, epochs=20,
batch_size=16):
    # [Code abbreviated for readability]

    # Calculate class weights to handle imbalance
    class_weights = {i: total_samples / (len(class_counts) * count) for i,
count
                    in enumerate(class_counts)}

    # Define model architecture for better class separation
    model = Sequential([
        # Embedding
        Embedding(max_words, embedding_dim),

        # Bidirectional LSTM for better context capture
        Bidirectional(LSTM(32, return_sequences=True)),

        # Global max pooling to capture the most important features
        GlobalMaxPooling1D(),

        # Dropout for regularization
        Dropout(0.3),

        # Hidden layer for better discrimination between classes
        Dense(64, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),
        Dropout(0.3),

        # Output layer
        Dense(num_classes, activation='softmax')
    ])
```

Key Features:

- Bidirectional LSTM architecture to capture context in both directions
- Class weights to handle imbalanced data distribution
- Custom threshold logic during prediction for better minority class detection
- Word embeddings to represent semantic relationships between words
- Regularization techniques (dropout, L2) to prevent overfitting

Naive Bayes Model

```
def train_naive_bayes(self, preprocessor=None, file_path=None):
    # [Code abbreviated for readability]

    # Calculate class weights to handle imbalance
    class_weights = compute_class_weight(
        class_weight='balanced',
        classes=np.unique(y_train),
        y=y_train
    )

    # Train the model with hyperparameter tuning
    param_grid = {
        'alpha': [0.01, 0.1, 0.5, 1.0, 2.0],
        'fit_prior': [True, False]
    }

    nb_model = MultinomialNB()
    grid_search = GridSearchCV(nb_model, param_grid, cv=5,
scoring='f1_weighted')
    grid_search.fit(X_train_vec, y_train)
```

Key Features:

- Optimized for sparse text feature representation using TF-IDF
- Smoothing implemented via alpha parameter tuning
- Uses weighted F1 scoring for optimization to handle class imbalance
- Maintains all components (model, vectorizer, encoder) for reproducible predictions

Bayesian Networks

We implemented two variants of Bayesian Networks: a traditional approach with n-gram features and another using Word2Vec embeddings:

```python
def train_bayesian_network(self, text_column, label_column, k_features=90):
    # [Code abbreviated for readability]

    # 1. Build the CountVectorizer and get the n-gram counts
    self.cv = CountVectorizer(ngram_range=(1, 2), stop_words='english')
    X_counts = self.cv.fit_transform(text_column)

    # 2. Select top k n-grams using a chi-square test
    self.selector = SelectKBest(chi2, k=k_features)
    X_selected = self.selector.fit_transform(X_counts, y)

    # 3. Build a feature DataFrame
    X_features = (X_selected > 0).astype(int)
    df_features = pd.DataFrame(X_features.toarray(),
columns=self.selected_features)

    # 4. Learn the Bayesian Network structure and parameters
    hc = HillClimbSearch(df_features)
    best_structure = hc.estimate(scoring_method=BicScore(df_features))
    self.trained_model = BayesianNetwork(best_structure.edges())
    self.trained_model.fit(df_features, estimator=BayesianEstimator,
prior_type='BDeu')
```

Word2Vec-based Bayesian Network:

```python
def train_bayesian_network_W2V(self, text_column, label_column,
vector_size=90,
                              window=5, min_count=1, n_bins=35):
    # [Code abbreviated for readability]

    # 1. Tokenize the text: split each sentence into words.
    sentences = [text.split() for text in text_column]

    # 2. Train a Word2Vec model on these tokenized sentences.
    self.w2v_model = Word2Vec(
        sentences,
        vector_size=vector_size,
        window=window,
        min_count=min_count,
        workers=4
    )
    # 3. Compute sentence embeddings for each text sample.
    X_w2v = np.array([sentence_embedding(sent, self.w2v_model) for sent in
sentences])

    # 4. Discretize the continuous embeddings.
    self.discretizer = KBinsDiscretizer(
        n_bins=n_bins,
        encode='ordinal',
        strategy='quantile'
    )
```

Key Features:

- Structure learning with Hill Climbing and BIC score optimization
- Feature selection with chi-square tests for traditional approach
- Word embeddings with discretization for the Word2Vec variant
- Variable elimination for efficient inference
- Effective handling of feature dependencies

Hidden Markov Model

```
def train_hidden_markov_model(self, X_train, y_train, n_components=2,
random_state=42):
    # [Code abbreviated for readability]

    # Create and train separate HMM for each sentiment class
    hmm_models = {}

    for sentiment in range(n_labels):
        # Filter data for current sentiment
        X_sentiment = X_train_scaled[y_train == sentiment]

        # Train HMM for this sentiment
        sentiment_model = hmm.GaussianHMM(
            n_components=n_components,
            covariance_type="full",
            n_iter=200,
            random_state=random_state
        )

        # Fit the model
        sentiment_model.fit(X_sentiment)
        hmm_models[sentiment] = sentiment_model
```

Key Features:

- Separate models for each sentiment class
- Class-conditional sequence modeling
- Gaussian emission probabilities for continuous features
- Log-likelihood scoring for classification
- Multiple hidden states to capture text patterns

Results and Comparative Analysis

Performance Overview

Our models demonstrated varying performance characteristics on the test dataset:

Model	Accuracy
Neural Network	85.37%
Naive Bayes	84.05%
Decision Tree	81.79%
Bayesian Network	75.14%
Hidden Markov Model	32.73%

Detailed Model Performance Metrics

Neural Network (RNN)

	precision	recall	f1-score
Class 0	0.94	0.88	0.91
Class 1	0.77	0.89	0.83
Class 2	0.88	0.79	0.83
accuracy			0.85
macro avg	0.86	0.85	0.85
weighted avg	0.86	0.85	0.85

Naive Bayes

	precision	**recall**	**f1-score**
Class 0	0.92	0.88	0.90
Class 1	0.73	0.93	0.82
Class 2	0.92	0.72	0.81
accuracy			0.84
macro avg	0.86	0.84	0.84
weighted avg	0.86	0.84	0.84

Decision Tree

	precision	**recall**	**f1-score**
Class 0	0.94	0.83	0.88
Class 1	0.69	0.94	0.80
Class 2	0.90	0.68	0.78
accuracy			0.82
macro avg	0.85	0.82	0.82
weighted avg	0.84	0.82	0.82

Bayesian Network

	precision	**recall**	**f1-score**
Class 0	0.95	0.73	0.83
Class 1	0.61	0.90	0.73
Class 2	0.83	0.62	0.71
accuracy			0.75
macro avg	0.80	0.75	0.75
weighted avg	0.80	0.75	0.75

Hidden Markov Model

	precision	**recall**	**f1-score**
Class 0	0.42	0.02	0.04

	precision	recall	f1-score
Class 1	0.00	0.00	0.00
Class 2	0.33	0.97	0.49
accuracy			0.33
macro avg	0.25	0.33	0.18
weighted avg	0.25	0.33	0.18

Comparative Analysis

1. **Neural Network (RNN)** achieved the highest overall accuracy at 85.37%. It shows strong balanced performance across all three classes with particularly good precision for negative (Class 0) and positive (Class 2) sentiments at 0.94 and 0.88 respectively. The model demonstrates its ability to effectively learn sequential patterns in text data.
2. **Naive Bayes** performed nearly as well with 84.05% accuracy. It shows excellent precision for negative and positive classes (0.92 for both) but lower precision for neutral content (0.73), suggesting it sometimes misclassifies other sentiments as neutral. The high recall for neutral content (0.93) indicates it captures most instances of this class.
3. **Decision Tree** achieved a solid 81.79% accuracy with similar patterns to Naive Bayes. It shows high precision for negative and positive classes (0.94 and 0.90) but struggles with neutral precision (0.69). However, it has the highest recall for neutral content (0.94), making it effective at identifying this class.
4. **Bayesian Network** reached 75.14% accuracy with an interesting pattern: it has the highest precision for negative sentiment (0.95) but lower recall (0.73), suggesting it's very selective about what it classifies as negative. It shows a bias toward classifying content as neutral (0.90 recall) but with lower precision (0.61).
5. **Hidden Markov Model** performed poorly with only 32.73% accuracy. The confusion matrix shows it classified almost everything as positive (Class 2), with 97% recall but only 33% precision for this class. It completely failed to identify neutral content (Class 1) with 0% for all metrics, suggesting fundamental issues with the model's ability to capture relevant patterns for sentiment classification.
6. **Class Imbalance Effects**: All models except HMM show some bias toward the neutral class (Class 1) with higher recall but lower precision, reflecting the class distribution in the training data. Neural Networks and Naive Bayes managed this challenge better than the others.

Conclusion

Phase 1 Achievements

- Successfully implemented five different model types for sentiment analysis
- Achieved strong performance with Neural Networks and Naive Bayes models
- Identified strengths and weaknesses of different approaches for the sentiment classification task
- Demonstrated effective preprocessing and feature engineering techniques specific to each model

Challenges and Limitations

- Hidden Markov Model performed poorly, suggesting it may not be appropriate for this type of classification task
- Short headline text provides limited context for sentiment analysis
- Class imbalance remains a challenge for all models to varying degrees
- Efficient feature representation differs substantially across models
- Some models struggle with capturing semantic relationships in short text

References

- Scikit-learn documentation for model implementations
- PGMPY documentation for graphical models
- TensorFlow documentation for neural network implementation
- Course lecture materials on ML algorithms