lab5_text_classification

Harito ID

2025-10-23

Lab 5: Text Classification

Objective

To build a complete text classification pipeline, from raw text to a trained machine learning model, using the tokenization and vectorization techniques learned in previous labs.

Theory

Text Classification is the task of assigning categories or labels to text documents. Common applications include sentiment analysis, spam detection, and topic labeling.

We will use a **supervised learning** approach, meaning we will train our model on a dataset where each text document is already associated with a known label.

Pipeline: Raw Text -> Tokenization -> Vectorization -> Machine Learning Model -> Prediction

Model: We will use **Logistic Regression**, a simple yet effective linear model for binary classification.

Evaluation: To assess our model's performance, we will use metrics such as Accuracy, Precision, Recall, and F1-score.

Task 1: Data Preparation (with scikit-learn)

1. **Dataset:** We'll use a small, in-memory dataset for simplicity.

   ```
   texts = [
       "This movie is fantastic and I love it!",
       "I hate this film, it's terrible.",
       "The acting was superb, a truly great experience.",
       "What a waste of time, absolutely boring.",
       "Highly recommend this, a masterpiece.",
       "Could not finish watching, so bad."
   ]
   labels = [1, 0, 1, 0, 1, 0] # 1 for positive, 0 for negative
   ```

2. **Vectorize:** Use your `TfidfVectorizer` (or `CountVectorizer`) from Lab 3 (or Lab 2) to transform these texts into numerical features.

Task 2: TextClassifier Implementation

1. **Create the file:** `src/models/text_classifier.py`.
2. **Implement the `TextClassifier` class:**
   - The constructor `__init__(self, vectorizer: Vectorizer)` should accept a `Vectorizer` instance.
   - It should have an attribute `_model` to store the trained `LogisticRegression` model from `scikit-learn`.
3. **Implement `fit(self, texts: List[str], labels: List[int])`:**
   - This method will train the classifier.
   - First, use the `vectorizer` to `fit_transform` the input `texts` into a feature matrix `X`.
   - Initialize a `LogisticRegression` model (e.g., `solver='liblinear'` for small datasets).
   - Train the model using `model.fit(X, labels)`.
4. **Implement `predict(self, texts: List[str]) -> List[int]`:**
   - This method will make predictions on new texts.
   - First, use the `vectorizer` to `transform` the input `texts` into a feature matrix `X`.
   - Use the trained `_model` to predict labels: `_model.predict(X)`.
5. **Implement `evaluate(self, y_true: List[int], y_pred: List[int]) -> Dict[str, float]`:**
   - This method will calculate evaluation metrics.
   - Use `sklearn.metrics` functions (`accuracy_score`, `precision_score`, `recall_score`, `f1_score`) to compute and return a dictionary of metrics.

Task 3: Evaluation

- Create a new test file: `test/lab5_test.py`.
- Define the `texts` and `labels` dataset.
- Split the data into training and testing sets (e.g., 80% train, 20% test). You can use `sklearn.model_selection.train_test_split`.
- Instantiate your `RegexTokenizer` and `TfidfVectorizer`.
- Instantiate your `TextClassifier` with the vectorizer.
- Train the classifier using the training data.
- Make predictions on the test data.
- Evaluate the predictions and print the metrics.

Advanced Example: Sentiment Analysis with PySpark

To handle larger datasets that don't fit into a single machine's memory, we can use Apache Spark. The file `test/lab5_spark_sentiment_analysis.py` provides an example of how to build a text classification pipeline using PySpark.

Code Analysis

The source code uses a Spark ML `Pipeline` to process the data and train the model. Here are the main steps:

1. **Initialize Spark Session:**

   ```
   spark = SparkSession.builder.appName("SentimentAnalysis").getOrCreate()
   ```

This is the entry point for any Spark application.

2. **Load Data:** Data is read from the `data/sentiments.csv` file. This file contains "text" and "sentiment" columns.

   ```
   df = spark.read.csv(data_path, header=True, inferSchema=True)
   # Convert -1/1 labels to 0/1: Normalize sentiment labels
   df = df.withColumn("label", (col("sentiment").cast("integer") + 1) / 2)
   # Drop rows with null sentiment values before processing
   initial_row_count = df.count()
   df = df.dropna(subset=["sentiment"])
   ```

3. **Build Preprocessing Pipeline:** A `Pipeline` in Spark ML consists of a sequence of `Transformer` and `Estimator`.

   - **`Tokenizer`**: Splits text into words (tokens).

     ```
     tokenizer = Tokenizer(inputCol="text", outputCol="words")
     ```

   - **`StopWordsRemover`**: Removes common stop words from the token list.

     ```
     stopwordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered_words
     ```

   - **`HashingTF`**: Converts a set of tokens into a fixed-size feature vector using a hashing technique. This is an efficient way to vectorize text.

     ```
     hashingTF = HashingTF(inputCol="filtered_words", outputCol="raw_features", numF
     ```

   - **`IDF` (Inverse Document Frequency)**: Rescales the feature vectors produced by `HashingTF`. It down-weights terms that appear frequently in the corpus.

     ```
     idf = IDF(inputCol="raw_features", outputCol="features")
     ```

4. **Train the Model:**

   - **`LogisticRegression`**: The model used for classification.

     ```
     lr = LogisticRegression(maxIter=10, regParam=0.001, featuresCol="features", lab
     ```

   - **Assemble the Pipeline**: All steps are combined into a single `Pipeline`.

     ```
     pipeline = Pipeline(stages=[tokenizer, stopwordsRemover, hashingTF, idf, lr])
     ```

   - **Training**: Call `pipeline.fit()` on the training data. Spark will automatically execute all stages in the pipeline.

     ```
     model = pipeline.fit(trainingData)
     ```

5. **Evaluate the Model:**

   - Use `model.transform()` on the test data to get predictions.
   - `MulticlassClassificationEvaluator` is used to calculate metrics like `accuracy` and `f1`.

     ```
     evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
     accuracy = evaluator.evaluate(predictions)
     ```

Running the Code

To run this file, you need to have `pyspark` installed. Then, you can run it as a regular Python script:

```
python test/lab5_spark_sentiment_analysis.py
```

Task 4: Evaluating and Improving Model Performance

After getting an initial result, a crucial question is: “How can we improve the model?” Here are some common approaches:

1. Improve Preprocessing and Feature Selection

The quality of the input features directly impacts model performance.

- **Noise Filtering:** Remove special characters, URLs, HTML tags, or other meaningless words.
- **Vocabulary Reduction:** Limit the vocabulary to include only words that appear with a certain frequency (e.g., remove words that are too rare or too common). This helps reduce the dimensionality of the feature space and combat noise.
- **Reduce TF-IDF Dimensionality:** When using `HashingTF` in Spark, you can experiment with different `numFeatures` values. A smaller number of features might help the model generalize better if the data is noisy.

2. Use Advanced Embedding Methods

Instead of TF-IDF, we can use dense embeddings to represent text. These embeddings often capture the semantics of words better.

- **Word2Vec:** Train a Word2Vec model on your text corpus to generate word vectors. You can then average the word vectors in a sentence to create a feature vector for the sentence. Spark ML provides a `Word2Vec` implementation.
- **Pre-trained Embeddings:** Use pre-trained embeddings like GloVe or FastText.

3. Experiment with More Complex Model Architectures

The `LogisticRegression` model is a good baseline, but more complex models can capture non-linear relationships in the data.

- **Naive Bayes:** A simple probabilistic model that often works well for text tasks.
- **Gradient-Boosted Trees (GBTs):** A powerful ensemble model that can yield high performance.
- **Neural Networks:** These models can learn hierarchical representations of text and often give the best results on large datasets.

Students are encouraged to experiment with these methods and compare the results to better understand the factors that influence the performance of a text classification system.