Quick Sentiments: Streamlining Your NLP Pipeline

The quick_sentiments package is designed to provide an efficient, end-to-end solution for sentiment analysis, abstracting away much of the complexity of traditional NLP workflows. With just a few intuitive commands, you can preprocess your text data, build and train a machine learning pipeline, and generate predictions on new, unseen data.

This document will walk you through the three primary steps to leverage the power of quick_sentiments:

1. **Text Preprocessing**: Cleaning and preparing your raw text data.
2. **Pipeline Execution**: Building and training your chosen vectorization and machine learning model.
3. **Prediction Generation**: Applying the trained model to new data.

Let's dive into each command in detail.

1. Data Preprocessing with pre_process()

Before any machine learning model can understand human language, text data needs to be cleaned and normalized. The pre_process() function is your package's robust tool for this crucial step. It takes raw text and applies a series of configurable cleaning transformations to prepare it for vectorization.

In thisexample, *pre_process()* is applied to a column of your DataFrame:

df_train = df_train.with_columns(

pl.col(response_column).map_elements(lambda x: pre_process(x, remove_brackets=True)).alias("processed")

)

What's Happening Here:

- df_train.with_columns(...): This Polars (or Pandas, if pl is aliased to pd) method is used to add a new column to your DataFrame.
- pl.col(response_column): Selects the column containing your raw text data (e.g., "reviewText").
- .map_elements(lambda x: pre_process(x, remove_brackets=True)): Applies the pre_process function to each element (x) in the response_column.
    - **pre_process(doc, ...)**: This function handles a wide array of text cleaning operations, including:

- Removing text within square brackets (remove_brackets=True).
- Removing URLs and email addresses.
- Stripping HTML tags.
- Converting text to lowercase.
- Tokenization (breaking text into words).
- Removing punctuation from individual tokens.
- Removing common English stop words (e.g., "the", "a", "is").
- Lemmatization (reducing words to their base form, e.g., "running" to "run").
- Normalizing Unicode characters (e.g., converting accented characters to their ASCII equivalents).
- Removing extra spaces.

- .alias("processed"): Names the new cleaned text column "processed". This new column will contain the cleaned version of your original text, ready for the next stage.

2. Building and Training the Pipeline with run_pipeline()

The run_pipeline() function is the core orchestrator of your sentiment analysis. It takes your prepared data, applies the chosen text vectorization method, trains a specified machine learning model, and evaluates its performance. It intelligently handles the entire training workflow.

```
dt = run_pipeline(
    vectorizer_name="wv", # BOW, tf, tfidf, wv
    model_name="logit", # logit, rf, XGB
    df=df_train,
    text_column_name="processed",
    sentiment_column_name = "sentiment",
    perform_tuning = False
)
```

Input Variables for run_pipeline():

- **vectorizer_name** (str): Specifies the text vectorization technique to use. This determines how your text is converted into numerical features that an ML model can understand.

    - "BOW": Bag-of-Words (counts word occurrences).
    - "tf": Term Frequency (normalized word counts).
    - "tfidf": Term Frequency-Inverse Document Frequency (weights words by importance).
    - "wv": Word Embeddings (e.g., Word2Vec, capturing semantic meaning).
- **model_name** (str): Specifies the machine learning algorithm to train for sentiment classification. Three methods are available right now
    - "logit": Logistic Regression.
    - "rf": Random Forest.
    - "XGB": XGBoost (eXtreme Gradient Boosting).
- **df** (DataFrame): The input DataFrame containing both your preprocessed text and the corresponding sentiment labels. This is typically the df_train DataFrame after the pre_process step.
- **text_column_name** (str): The name of the column in df that contains the preprocessed text data (e.g., "processed" from the previous step).
- **sentiment_column_name** (str): The name of the column in df that contains the true sentiment labels (e.g., "sentiment").
- **perform_tuning** (bool): If True, the function will perform hyperparameter tuning (e.g., using GridSearchCV) for the chosen model, which can improve performance but takes significantly longer and may consume more memory for large datasets. If False, the model will be trained with default parameters.

Return Value of run_pipeline() (Stored in dt):

The run_pipeline() function returns a dictionary (dt) containing all the essential objects and metrics from the training process. This dictionary is crucial because it provides everything you need to make future predictions on new data without re-training.

{

"model_object": trained_model_object, # The trained machine learning model (e.g., LogisticRegression instance)

"vectorizer_name": vectorizer_name, # The name of the vectorizer used (e.g., "wv")

"vectorizer_object": fitted_vectorizer_object, # The fitted vectorizer object (e.g., Word2Vec model, TfidfVectorizer)

"label_encoder": label_encoder, # The fitted LabelEncoder object, used to convert sentiment labels to numerical format and back

"y_test": y_test, # The true sentiment labels from the test set

"y_pred": y_pred, # The predicted sentiment labels on the test set

"accuracy": accuracy_score(y_test, y_pred), # The accuracy score of the model on the test set

"report": classification_report(y_test, y_pred, output_dict=True, target_names=label_encoder.classes_) # Detailed classification report (precision, recall, f1-score)

}

3. Making Predictions with make_predictions()

Once your pipeline is trained and evaluated using run_pipeline(), the make_predictions() function allows you to apply this trained model to new, unseen text data. It seamlessly integrates the vectorizer and model objects returned by run_pipeline() to generate sentiment predictions.

```
make_predictions(
    new_data=new_data,
    text_column_name="processed",
    vectorizer=dt["vectorizer_object"],
    best_model=dt["model_object"],
    label_encoder=dt["label_encoder"],
    prediction_column_name="sentiment_predictions"
)
```

What's Happening Here:

The make_predictions() function takes your new data, preprocesses it (if needed), vectorizes it using the *same* fitted vectorizer from training, and then uses the *trained* model to predict sentiments.

Input Variables for make_predictions():

- **new_data** (DataFrame): The DataFrame containing the new text data on which you want to make predictions. This DataFrame should ideally have the same structure as your training data, particularly the text column.
- **text_column_name** (str): The name of the column in new_data that contains the raw or preprocessed text. It's crucial that this text is prepared in the same way as your training text (e.g., using pre_process()).
- **vectorizer** (object): This is the *fitted vectorizer object* obtained directly from the run_pipeline() output (dt["vectorizer_object"]). Using the same fitted vectorizer ensures that new text is transformed into features consistently with how the model was trained.
- **best_model** (object): This is the *trained machine learning model object* obtained directly from the run_pipeline() output (dt["model_object"]). This is the model that will perform the actual sentiment classification.
- **label_encoder** (object): This is the *fitted LabelEncoder object* obtained directly from the run_pipeline() output (dt["label_encoder"]). It's used to convert the numerical predictions from the model back into human-readable sentiment labels (e.g., 0 -> "Negative", 1 -> "Positive").
- **prediction_column_name** (str, optional): An optional custom name for the new column that will be added to new_data containing the predicted sentiment labels. If not provided, a default name will be used.

The make_predictions() function will typically return the new_data DataFrame with an additional column containing the predicted sentiments.

By leveraging these three streamlined commands, the quick_sentiments package empowers you to quickly and effectively build, train, and deploy sentiment analysis models with minimal boilerplate code.