NLP for Sentiment Analysis with Logistic Regression

Note:We highly recommend trying this tutorial on our webpage at (https://dev-aligator.github.io/CS221/) for an enhanced experience. You'll benefit from Vietnamese language support, live demonstrations, and an improved user interface for more enjoyable experiments.

In this tutorial, you will learn how to perform aspect category sentiment analysis on Vietnamese restaurant reviews datasets (https://drive.google.com/file/d/1yjZ0sDD2kAKOZK78MFqWJyaDcCSxnIqN/view?usp=drive_link) using Logistic Regression with Gradient Descent.

Aspect category sentiment analysis is the process of classifying text data, such as customer reviews or social media comments, into specific aspect categories (e.g., "FOOD#QUALITY," "SERVICE#GENERAL") and determining the sentiment associated with each aspect (e.g., positive, negative, neutral).

Instead of employing separate models for each entity, we will implement a single model capable of classifying sentiments across all aspect categories efficiently."

In this notebook, Our goal is to build a robust sentiment analysis model for Vietnamese text data. We will walk through the following key steps:

1. Examine and understand the data
2. Feature Extraction: We use the TF (Term Frequency) vectorization technique to convert the preprocessed text data into numerical features suitable for machine learning.
3. Model Training
    - We employ a logistic regression classifier, designed to predict sentiments for different aspect categories concurrently
    - This model has the capability to classify text into specific aspect categories and assign sentiments such as negative, neutral, or positive.
4. Model Evaluation

```
import numpy as np
import pandas as pd
import tensorflow as tf
import torch
import matplotlib.pyplot as plt
tf.get_logger().setLevel('ERROR')
```

```
2023-10-08 23:27:29.061168: I tensorflow/core/util/port.cc:111] oneDNN custom operations are on. You may see slightly d
```

Data preparation

Data download

In this tutorial, you will use a dataset containing several thousand reviews of restaurant in Vietnamese. Download (https://drive.google.com/uc?export=download&id=1yjZ0sDD2kAKOZK78MFqWJyaDcCSxnIqN) and extract a zip file containing the csv files, or use the original txt files provided by VLSP2018 and follow this (https://github.com/Dev-Aligator/CS221/blob/master/Solution/csvConverter.py) to convert it to csv .

```
TRAIN_PATH = 'VLSP2018-SA-train-dev-test/csv/train.csv'
VAL_PATH = 'VLSP2018-SA-train-dev-test/csv/dev.csv'
TEST_PATH = 'VLSP2018-SA-train-dev-test/csv/test.csv'
```

Data loading

To load and process the dataset, we first define a function to read CSV files using the Pandas (https://pandas.pydata.org/docs/) library. The goal is to extract the review text and corresponding labels for sentiment analysis. Here's how we do it:

```
def read_csv(url):
  df = pd.read_csv(url)
  X = df.pop('review')
  y = df.replace({np.nan: 0,
                  'negative': 1,
                  'neutral': 2,
                  'positive': 3}).astype(np.uint8)
  print('X.shape:', X.shape, 'y.shape:', y.shape)
  return X, y
```

Now that we have the function to read and process data, let's apply it to the training, validation, and test datasets:

```
Xtrain, ytrain = read_csv(TRAIN_PATH)
Xdev,   ydev   = read_csv(VAL_PATH)
Xtest,  ytest  = read_csv(TEST_PATH)
```

```
X.shape: (2961,) y.shape: (2961, 12)
X.shape: (1290,) y.shape: (1290, 12)
X.shape: (500,) y.shape: (500, 12)
```

Define Aspects and Sentiments

Based on the VLSP dataset guidelines, we can determine that our analysis will have 12 distinct aspects, each associated with 3 different sentiments.

```
aspects = ['FOOD#PRICES',
            'FOOD#QUALITY',
            'FOOD#STYLE&OPTIONS',
            'DRINKS#PRICES',
            'DRINKS#QUALITY',
            'DRINKS#STYLE&OPTIONS',
            'RESTAURANT#PRICES',
            'RESTAURANT#GENERAL',
            'RESTAURANT#MISCELLANEOUS',
            'SERVICE#GENERAL',
            'AMBIENCE#GENERAL',
            'LOCATION#GENERAL']

sentiments = ['-', 'o', '+']    # Negative, Neutral, Positive
```

Next, we'll define a function to convert multi-output data into binary multi-label format.

```
def mo2ml(y):
  newcols = [f'{a} {s}' for a in aspects for s in sentiments]

  nrows, ncols = len(y), len(newcols)
  ml = pd.DataFrame(np.zeros((nrows, ncols), dtype='bool'),
                    columns=newcols)

  for i, a in enumerate(aspects):
      for j in range(1, 4):
          indices = y[a] == j
          ml.iloc[indices, i * 3 + j - 1] = True

  return ml
```

```
FOOD#QUALITY = 2 ===> FOOD#QUALITY - = 0 FOOD#QUALITY o = 0 FOOD#QUALITY + = 1
```

Convert our data to binary multi-label format using the `mo2ml` function

```
ytrain_ml = mo2ml(ytrain)
ydev_ml = mo2ml(ydev)
ytest_ml = mo2ml(ytest)
```

	FOOD#PRICES -	FOOD#PRICES o	FOOD#PRICES +	FOOD#QUALITY -	FOOD#QUALITY o	FOOD#QUALITY +
0	False	False	False	False	True	False
1	False	True	False	False	False	True
2	False	False	True	False	False	True
3	False	False	False	False	False	True
4	False	False	False	False	False	True
...	...	...	...	...	...	...
2956	False	False	False	False	False	True
2957	False	True	False	False	False	True
2958	False	False	False	False	False	False
2959	False	False	False	False	False	False
2960	False	False	False	False	False	True

2961 rows × 36 columns

We also need to ensure data is in the DataFrame format, converting it if necessary.

```
def mo2df(y):
  if isinstance(y, pd.DataFrame):
      return y
  return pd.DataFrame(y, columns=aspects)
```

Show a bar plot showing the values for each aspects from the training set:

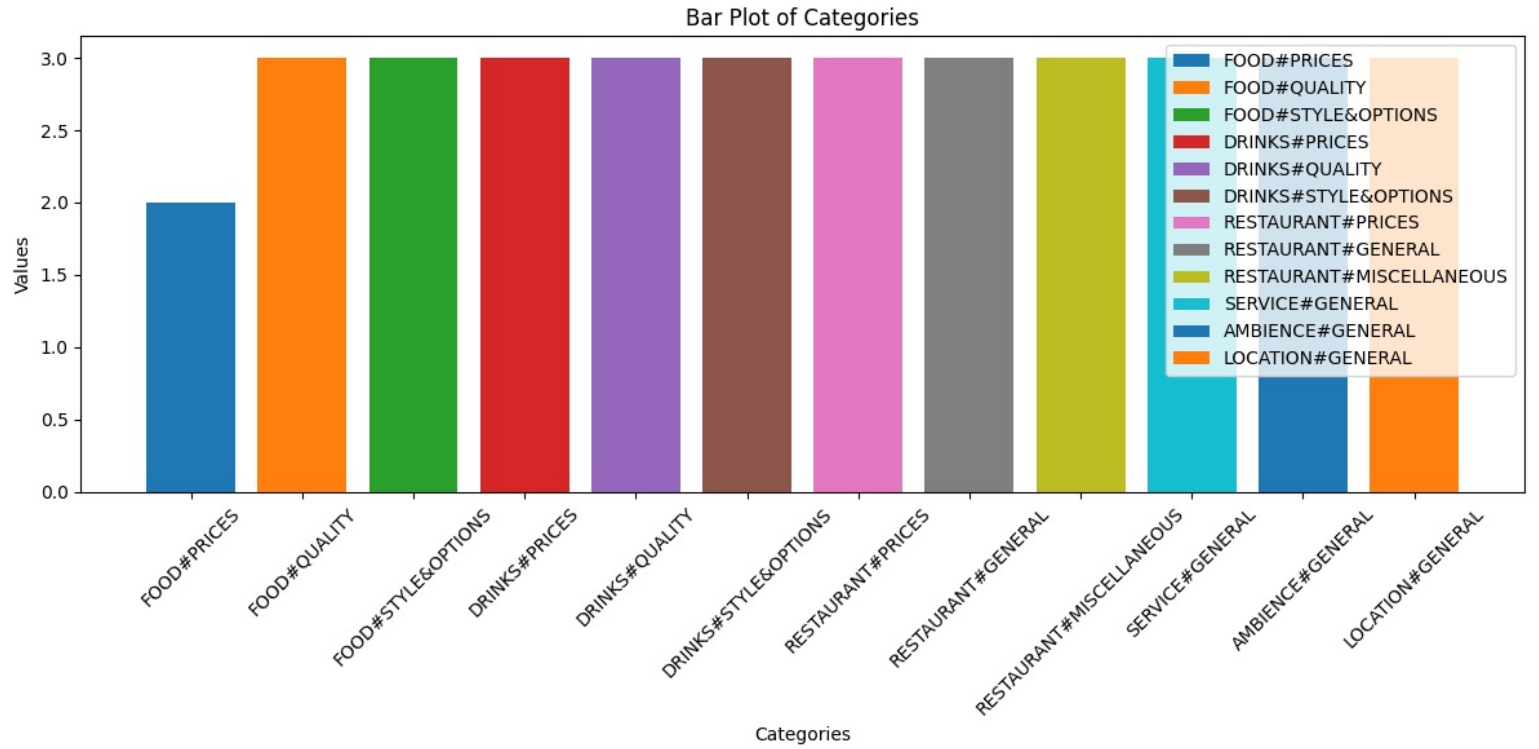
```
ytrain_transposed = ytrain.T
fig, ax = plt.subplots(figsize=(12, 6))

# Iterate through each category and plot the values
for category_index, category_name in dict(enumerate(aspects)).items():
    ax.bar(category_name, ytrain_transposed[category_index], label=category_name)

# Set labels, legend, and title
ax.set_xlabel('Categories')
ax.set_ylabel('Values')
ax.set_title('Bar Plot of Categories')
ax.legend()

# Rotate x-axis labels for better visibility
plt.xticks(rotation=45)

# Show the plot
plt.tight_layout()
plt.show()
```



Feature extraction

In this step, we convert our text data into term frequency features using CountVectorizer from scikit-learn.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(ngram_range=(1, 3), min_df=2, max_df=0.9)
```

Note:`ngram_range`:We define the range of word combinations (unigrams, bigrams, and trigrams) to capture more context in the text.`min_df=2`: specifies that a term must appeart in at least 2 documents to be considered as a feature.`max_df=0.9`: specifies that if a term must appeart in more than 90% of the documents in the dataset are excluded.

Transformation

```
xtrain_baseCV = vectorizer.fit_transform(Xtrain)
xdev_baseCV = vectorizer.transform(Xdev)
xtest_baseCV = vectorizer.transform(Xtest)
```

Let's see the result after the transformation step.

```
sample_vector = xtrain_baseCV[0]

# Retrieve terms and their indices from the vocabulary
vocab_terms = {index: term for term, index in vectorizer.vocabulary_.items()}

# For each non-zero entry in the sample vector, print the term and its frequency value
for index in sample_vector.indices:
    print(f"Term: {vocab_terms[index]}, CV value: {sample_vector[0, index]}")
```

Model training

Evaluation functions

First, we create evaluation functions to check how well our model is doing. In this case, we'll make use of the 'classification_report' and 'f1 score'

```
from sklearn.metrics import f1_score, classification_report


def quick_f1(y_true, y_pred):
    y_pred = mo2ml(mo2df(y_pred))
    return round(f1_score(y_true, y_pred, average='micro', zero_division=0), 4)

def evaluate(model, X, y, average='micro'):
    yb_true  = mo2ml(y)

    yb_pred  = mo2df(model.predict(X))
    yb_pred  = mo2ml(yb_pred)

    return classification_report(yb_true, yb_pred, zero_division=0)
```

`The F1 score`: is the harmonic mean of precision and recall.
`Classification Report`: is a summary of various classification metrics for a machine learning model.

Set up & Training

Import the necessary libraries, including 'SGDClassifier' for stochastic gradient descent classification, `optuna` for hyperparameter optimization, `TPESampler` for the Tree-structured Parzen Estimator sampler, and `MultiOutputClassifier` (MOC) for multi-output classification.

```
from sklearn.linear_model import SGDClassifier
import optuna
from optuna.samplers import TPESampler
from sklearn.multioutput import MultiOutputClassifier as MOC
```

To simplify the challenging task of hyperparameter selection, we'll use Optuna. Now, we define a callback function for Optuna that tracks and saves the best model during the optimization process.

```
def callback(study, trial):
  if study.best_trial.number == trial.number:
    study.set_user_attr(key='best_model', value=trial.user_attrs['model'])
```

Next we define the objective function for Optuna. It contains the hyperparameters to optimize, including `class_weight` and `alpha`.

```
def logistic_objective(trial):
    params = dict(
        class_weight=trial.suggest_categorical('class_weight', ['balanced', None]),
        alpha=trial.suggest_float('alpha', 1e-7, 1e-2, log=True),  # Add alpha for L2 regularization.
        random_state=5,
    )
    # This function continues...
```

The `log` parameter in the `suggest_float` function suggests hyperparameters on a logarithmic scale.

Now create an instance of the MultiOutputClassifier (MOC) and train it using the SGDClassifier with the hyperparameters defined by the trial. The best model is saved as a user attribute for later reference.

```
    # logistic_objective continues here....

    clf = MOC(SGDClassifier(loss='log_loss', max_iter=200, **params))
    clf.fit(xtrain_baseCV, ytrain)
```

```
    trial.set_user_attr(key="model", value=clf)

    y_pred = clf.predict(xdev_baseCV)
    return quick_f1(ydev_ml, y_pred)
```

The choice of loss='log_loss' indicates that the logistic loss function (log loss) is used, making it equivalent to logistic regression.

Finally, set up the Optuna study with the specified sampler and optimization direction and run the optimization process by the `optimize`` method

```
sampler = TPESampler(seed=221)
logistic_study = optuna.create_study(sampler=sampler, direction='maximize')
logistic_study.optimize(logistic_objective, n_trials=50, callbacks=[callback])
```

TPESampler is one of the available samplers in Optuna. It uses a Bayesian optimization strategy to explore the hyperparameter search space efficiently.
n_trials=50 specifies that Optuna will perform 50 trials to optimize hyperparameters.

We evaluate the best model on the test dataset, examine its performance on training and development data, and review the selected hyperparameters."

```
clf = logistic_study.user_attrs['best_model']

print(evaluate(clf, xtest_baseCV, ytest))

print('train:', quick_f1(ytrain_ml, clf4.predict(xtrain_baseCV)))
print('dev: ', quick_f1(ydev_ml , clf4.predict(xdev_baseCV)))
print('test:', quick_f1(ytest_ml , clf4.predict(xtest_baseCV)))

print(clf.estimators_[0].get_params())
print(logistic_study.best_params)
```

```
              precision    recall  f1-score   support

           0       0.25      0.04      0.06        28
           1       0.52      0.46      0.49       175
           2       0.50      0.59      0.54       128
           3       0.00      0.00      0.00        11
           4       0.48      0.26      0.33        43
           5       0.87      0.96      0.91       403
           6       0.00      0.00      0.00        16
           7       0.00      0.00      0.00        53
           8       0.74      0.92      0.82       334
           9       0.00      0.00      0.00         3
          10       0.00      0.00      0.00        45
          11       0.00      0.00      0.00        28
```

Here, we have successfully constructed a Logistic Regression Model for sentiment classification across various aspect categories. However as you can see from the evaluation score - 0.5821 for the test set, indicates room for improvement.

Optimization techniques

Data processing

Looking at the review data, we find emojis, different letter cases, special characters. To prepare the data properly, we will define a Text Cleaner class as below:

```
class TextCleanerBase(BaseEstimator, TransformerMixin):
  def __init__(self):
    super().__init__()
```

```
    # Create preprocessing function
    self.normalize_unicode = partial(unicodedata.normalize, 'NFC')

    def fit(self, X, y=None):
      return self

    def transform(self, X):
      if not isinstance(X, pd.Series):
        X = pd.Series(X)

      return X.apply(str.lower)
              .apply(remove_emojis)
              .apply(self.normalize_unicode)

def remove_emojis(text):
  return demoji.replace(text, '')
```

You have the option to skip this step because its impact on model performance is minimal. Alternatively, you may explore another method to enhance accuracy.

Use TF-IDF instead of TF

Switching to TF-IDF from TF improves our model by assigning more weight to relevant words and reducing feature complexity, leading to more accurate and versatile sentiment classification.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(ngram_range=(1, 3), min_df=2, max_df=0.9)
```

After implementing the two methods mentioned above, we obtain the final result:

```
train: 0.9969
dev:   0.6629
test: 0.6224
{'alpha': 6.397620627963636e-05, 'average': False, 'class_weight': 'balanced', 'early_stopping': False, 'epsilon': 0.1,
{'class_weight': 'balanced', 'alpha': 6.397620627963636e-05}
```

This tutorial concludes here. We hope you have found it helpful. We highly recommend downloading the notebook and experimenting with your own optimization methods.

Last updated 2023-10-17 UTC.