hw01

March 6, 2020

HW1 Movie Review Sentiment Analysis

Due: March 2020 6th, 23:59

In this homework you will create different models to generate the positive/negative sentiment classification of movie reviews.

This homework should be done individually without cooperation with others.

You are given the following files: - `hw01.ipynb`: Notebook file with starter code - `train.txt`: Training set to train your model - `test.txt`: Test set to report your model's performance - `sample_prediction.csv`: Sample file your prediction result should look like - `utils/`: folder containing all utility code for the series of homeworks

Remember to leverage code in `utils/`, so you don't have to build everything from Scratch. - `load_data(filename)`: load the input data and return sklearn.Bunch object. For basic usage of Bunch object, please refer to sklearn documentation. - `save_prediction(arr, filename)`: save your prediction into the format required by the course

0. Install Anaconda

If you do not yet have Python and Jupyter Notebook on your laptop, use this link (https://www.anaconda.com/) to install anaconda. Anaconda is a suite that provides one stop solution for all you need for Python development environment. This site contains installation document for Windows, Mac, and Linux, choose the one that suits your operating system.

Tips: You may want to consider installing Jupyter Extensions (link: https://github.com/ipython-contrib/jupyter_contrib_nbextensions), and turn on extensions such as `ExcuteTime` and `Table of Contents(2)`. You may find them very helpful to assist you finishing homework. However, this is definitely not a necessary requirement.

1. Feature Dictionary Vectorization

A quite unique step for NLP is to engineer the raw text input into numerical features. You will eventually implement several featurizers, just like `dummy_featurize`, that distinguish your models with others. However, we are not there yet. For convenience we allow you to represent the features using a dictionary, look at the `dummy_featurize` code. So each data point can be translated into a dictionary. However later we have to translate this list of dictionaries into a homogeneous data structure. Therefore, you need to first implement the pipeline method in the `SentimetnClassifier` class. Look into the code comment for more details. To ensure your implementation works, we also provided some test code in the cell below.

2. Better featurizers

Have you finished the first step, you can run the model using the provided featurizer. See the performance is nearly as good as a donkey? No surprise! The `dummy_featurize` should have been named `really_dummy_featurize`. Now it is your turn to implement better featurizers. For this homework, you need to implement at least 3 distinguishable featurizer. Describe your features briefly in the write-up and include the accuracy of the model. No idea at all? Look at your lecture notes for inspiration. Still no clue? Why not start with Bag of words?

Note: Model performance is important but it's not the only thing we care about, your work will also be rewarded by your creativity.

3. Optional: Try different learning methods

All the work you have done so far are related to feature engineering, and the featurized data is trained using Logistic Regression. Try to use different learning methods to train the model and see if you achieve any difference in the performance. Discuss your findings in the write-up.

4. Deliverables (zip them all)

- pdf version of your final notebook
- Use the best model you trained, generate the prediction for test.txt, name the output file prediction.csv (Be careful: the best model in your training set might not be the best model for the test set).
- HW1_writeup.pdf: summarize the method you used and report their performance. If you worked on the optional task, add the discussion. Add a short essay discussing the biggest challenges you encounter during this assignment and what you have learnt.

(**You are encouraged to add the writeup doc into your notebook using markdown/html langauge, just like how this notes is prepared**)

HW1 Write up

Data Pre-processing:

It is good to check/see that both the training and dev dataset has balanced categories.

I used nltk tokenizer to tokenize the texts into list of tokens, and use WordNetLemmatizer to normalize the text.

I implemented the vectorizer algorithm from scratch, however, it turns out to be slower

Using the sklearn learn Pipeline and build-in method will speed up.

Feature Engineering:

I tried to use the following features:

- bag of words(unigram) : around 7200 (with at least 10 occurance)
- bigram :around 12300 (with at least 10 occurance)
- some handcrafted features like number of unique words,number of punctions, etc., I also tried to use the nltk tagging to get the number of nouns, adjectives, and verbs : about 8
- tf-idf (on the unigram or on the bigram), I implement the equation by hand:

The training set has sparsity(number of non-zero fraction) = 0.9%, so we need to sparse format to store it for efficiency.

It turns out that the handcrafted features and only bigram without tf-idf does not help to improve the accuracy of the model.

Learning method:

I used the following methods:

- Logistic regression with L2 regulazation (default)
- SVM
- RandomForestClassfier
- XGBoost

The Logistric regression is the "baseline" and actually it reaches good performance on training/dev set.

RandomForestClassfier is easy to overfit, with large accuracy on training set but poor performance on dev set, and XGBoost encounters the same problem, even with a reasonable number of trees and max depth of the trees.

SVM gives a good estimation on both training and dev set. However, they did not surpass the score of baseline model for the dev set.

The caveat for all the models is that they all need finer tuning.

To reach better accuracy, we might need more training data, or use other feature engineer techiques.

Result

My final feature selection with unigram and tf-idf applied improves the dev accuracy, and the prediction power mainly comes from unigram.

Please see the second-to-last Cell to see the comparasions.

It agrees with the paper(2015)

https://www.sciencedirect.com/science/article/pii/S1877050915001520

which says "We found that unigram is the best method to extract sentiment from the review." And they got best accuracy around 82% - 84%.

1 ================ Coding Starts Here ====================

```
In [2]: %load_ext autoreload
        %autoreload 2
        %matplotlib inline
        import os
        import sys
        import pandas as pd
        import numpy as np

        # add utils folder to path
        p = os.path.dirname(os.getcwd())
        if p not in sys.path:
            sys.path = [p] + sys.path
```

```
from utils.hw1 import load_data, save_prediction
```

1.1 Featurizer

```
In [3]: """
        !! Do not modify !!
        """
        def dumb_featurize(text):
            feats = {}
            words = text.split(" ")

            for word in words:
                if word == "love" or word == "like" or word == "best":
                    feats["contains_positive_word"] = 1
                if word == "hate" or word == "dislike" or word == "worst" or word == "awful":
                    feats["contains_negative_word"] = 1

            return feats
```

```
In [4]: from nltk.tokenize import word_tokenize
        from nltk.corpus import stopwords
        from nltk.stem.wordnet import WordNetLemmatizer
        from nltk.tag import pos_tag
        from nltk.tag.perceptron import PerceptronTagger
        import string

        tagger = PerceptronTagger()

        def better_featurize(text):
            feats = {}
            lemmatizer = WordNetLemmatizer()
            words = word_tokenize(text.lower())
            words = [lemmatizer.lemmatize(word) for word in words]

            stop_words = stopwords.words('english')

            def n_gram(n):
                #n - gram
                ngrams = []
                for i in range(len(words)-n+1):
                    g = '_'.join(words[i:i+n])
                    ngrams.append(g)
                return ngrams

            feats = Counter(words) + Counter(n_gram(2)) # merge uni-gram and bigram
            #feats = Counter(words)
            #feats = {k:v for k,v in feats.items() if k not in stop_words and k not in
        string.punctuation}
            #feats = {k:v/len(words) for k,v in feats.items()}
            #feats['num_unique_words'] = len(set(words))
            #feats['num_chars'] = len(text)
            #feats['num_stopwords'] =  len([word for word in words if word in
        stopwords.words('english')])
            #feats['num_punctions'] = len([word for word in words if word in
        string.punctuation])
            #feats['mean_word_len'] = np.mean([len(word) for word in words])
            #pos_list = pos_tag(words)
            #pos_list = tagger.tag(words)
            #feats['num_nouns'] = len([w for w in pos_list if w[1] in
        ('NN','NNP','NNPS','NNS')])
            #feats['num_adjs'] = len([w for w in pos_list if w[1] in ('JJ','JJR','JJS')])
            #feats['num_verbs'] = len([w for w in pos_list if w[1] in
        ('VB','VBD','VBG','VBN','VBP','VBZ')])
            return feats
```

1.2 Model Class

```
In [5]: from collections import Counter
        from scipy.sparse import dok_matrix
        from numpy import count_nonzero
        from math import log
        from sklearn.pipeline import Pipeline
        from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer,
        TfidfTransformer
        from sklearn.feature_extraction import DictVectorizer
        from sklearn.linear_model import LogisticRegression
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.ensemble import GradientBoostingClassifier
        from sklearn.svm import LinearSVC
        from sklearn.naive_bayes import MultinomialNB
        from sklearn.preprocessing import normalize
        import xgboost as xgb


        class SentimentClassifier:
            def __init__(self, feature_method=dumb_featurize, min_feature_ct=1, L2_reg=1.0):
                """
                :param feature_method: featurize function
                :param min_feature_count: int, ignore the features that appear less than this
        number to avoid overfitting
                """
                self.feature_vocab = {}
                self.feature_method = feature_method
                self.min_feature_ct = min_feature_ct
                self.L2_reg = L2_reg

            def featurize(self, X):
                """
                # Featurize input text

                :param X: list of texts
                :return: list of featurized vectors
                """
                featurized_data = []
                for text in X:
                    feats = self.feature_method(text)
                    featurized_data.append(feats)
                return featurized_data

            def pipeline(self, X, training=False):
                """
                Data processing pipeline to translate raw data input into sparse vectors
                :param X: featurized input
                :return X2: 2d sparse vectors

                Implement the pipeline method that translate the dictionary like feature vectors
        into
                homogeneous numerical vectors, for example:
                [{"fea1": 1, "fea2": 2},
                 {"fea2": 2, "fea3": 3}]
                 -->
                 [[1, 2, 0],
                  [0, 2, 3]]

                Hints:
                1. How can you know the length of the feature vector?
                2. When should you use sparse matrix?
                3. Have you treated non-seen features properly?
                4. Should you treat training and testing data differently?
                """
                # Have to build feature_vocab during training
                if training:
```

```
            print("Now training...")
            # the commented out part is to implement the algorithm from scratch,
however, it turns out to be slower
            # Using the scikit learn Pipeline and build-in method will speed up
            '''
            self.idf_ = {} # count the document which have the features
            for feature_dic in X:
                for feature, value in feature_dic.items():
                    self.feature_vocab[feature] = self.feature_vocab.get(feature, 0) +
value
                    self.idf_[feature] = self.idf_.get(feature, 0) + 1
            index = 0

            # sort the features by the occurance
            for k, v in sorted(self.feature_vocab.items(), key=lambda item: item[1],
reverse=True):
                if v < self.min_feature_ct:
                    del self.feature_vocab[k]
                    del self.idf_[k]
                else:
                    index += 1
                    idf = self.idf_[k]
                    self.idf_[k] = log((1+len(X))/(1+idf))+1
                    self.feature_vocab[k] = index
            self.feature_vocab['__UNKNOWN__'] = 0
        X2 = []
        # Translate raw texts into vectors
        for feature_dic in X:
            feature_vector = []
            for feature in self.feature_vocab.keys():
                if '_' in feature:
                    feature_vector.append(feature_dic.get(feature, 0))
                else: # unigram apply the tf-idf formula
                    feature_vector.append(bool(feature_dic.get(feature,
0))*self.idf_.get(feature, 0))

            X2.append(feature_vector)
            '''
            self.vec = DictVectorizer().fit(X)
            X2 = self.vec.transform(X).toarray()
            self.feature_sel = np.count_nonzero(X2,axis=0)>=self.min_feature_ct
            self.pipe = TfidfTransformer().fit(X2[:,self.feature_sel])
        #X2 = np.array(X2)
        X2 = self.vec.transform(X).toarray()
        X2 = X2[:,self.feature_sel]
        X2 = self.pipe.transform(X2).toarray()
        sparsity = count_nonzero(X2) / float(X2.size)
        # for sparse matrix, store them use the sparse format
        if (sparsity < 0.02):
            X2 = dok_matrix(X2)
        #X2 = normalize(X2, axis=1, norm='l1')
        return X2

    def fit(self, X, y):
        X = self.pipeline(self.featurize(X), training=True)

        D, F = X.shape
        print(D,F)
        self.model = LogisticRegression(C=self.L2_reg)
        #self.model = LinearSVC(C=self.L2_reg)
        #self.model = RandomForestClassifier(n_estimators=2000,
        #                                    max_depth=5)
        #self.model = xgb.XGBClassifier(n_estimators = 2500,
        #                               random_state = 2009,
        #                               max_depth = 5,
        #                               learning_rate= 0.05,
        #                               colsample_bytree = 0.2,
        #                               subsample = 0.8,
        #                               min_child_weight = 2.,
        #                               n_jobs=4)
        self.model.fit(X, y)

        return self

    def predict(self, X):
        X = self.pipeline(self.featurize(X))
        return self.model.predict(X)

    def score(self, X, y):
        X = self.pipeline(self.featurize(X))
        return self.model.score(X, y)

    # Write learned parameters to file
    def save_weights(self, filename='weights.csv'):
        weights = [["__intercept__", self.model.intercept_[0]]]
        for feat, idx in self.feature_vocab.items():
            weights.append([feat, self.model.coef_[0][idx]])

        weights = pd.DataFrame(weights)
        weights.to_csv(filename, header=False, index=False)

        return weights
```

```
In [6]: """
        Run this to test your model implementation
        """

        cls = SentimentClassifier()
        X_train = [{"fea1": 1, "fea2": 2}, {"fea2": 2, "fea3": 3}]

        X = cls.pipeline(X_train, True)
        assert X.shape[0] == 2 and X.shape[1] >= 3, "Fail to vectorize training features"

        X_test = [{"fea1": 1, "fea2": 2}, {"fea2": 2, "fea3": 3}]
        X = cls.pipeline(X_test)
        assert X.shape[0] == 2 and X.shape[1] >= 3, "Fail to vectorize testing features"

        X_test = [{"fea1": 1, "fea2": 2}, {"fea2": 2, "fea4": 3}]
        try:
            X = cls.pipeline(X_test)
            assert X.shape[0] == 2 and X.shape[1] >= 3
        except:
            raise Exception("Fail to treat un-seen features")

        print("Success!!")
```

```
Now training…
Success!!
```

1.3 Run your model

```
In [7]: """
        Run this cell to test your model
        """
        from sklearn.model_selection import train_test_split

        data = load_data("train.txt")
        X, y = data.text, data.target
        X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)
        cls = SentimentClassifier(feature_method=dumb_featurize)
        cls = cls.fit(X_train, y_train)
        print("Training set accuracy: ", cls.score(X_train, y_train))
        print("Dev set accuracy: ", cls.score(X_dev, y_dev))
```

```
Now training…
7000 2
Training set accuracy:  0.526
Dev set accuracy:  0.5176666666666667
```

```
In [12]: """
         Run this cell to save weights and the prediction
         """
         try:
             weights = cls.save_weights()
         except:
             pass
         X_test = load_data("test.txt").text
         save_prediction(cls.predict(X_test))
```

1.4 Example of better featurizer

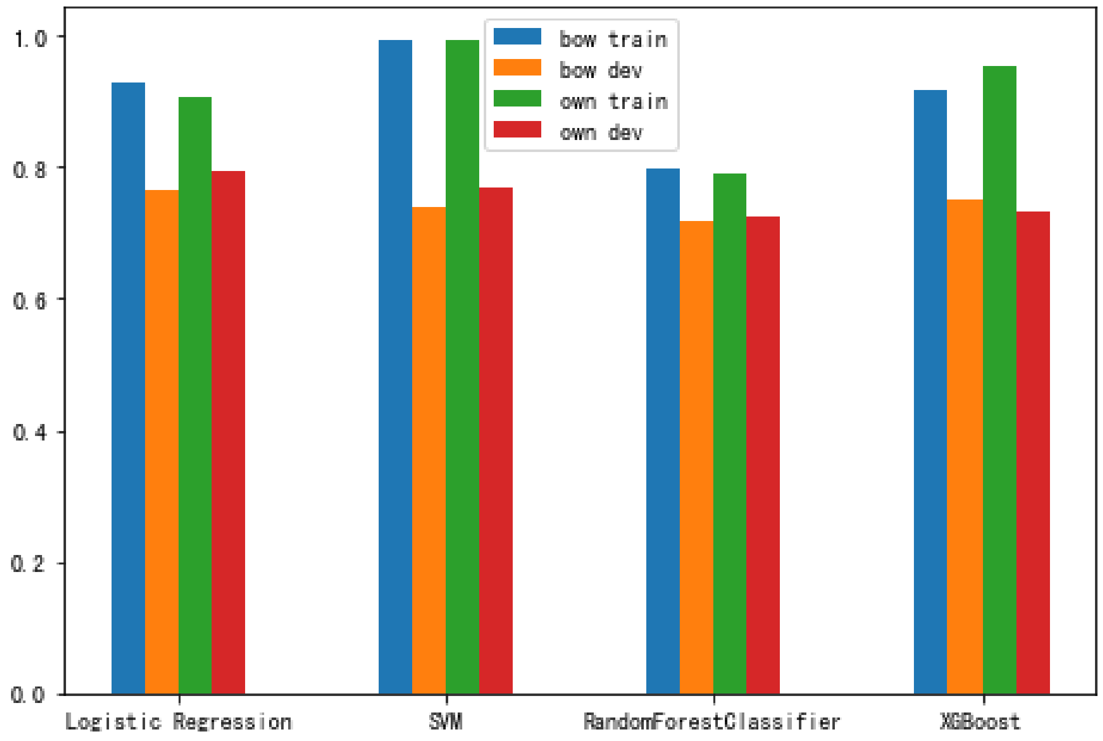
```
In [24]: def bag_of_words(text):
             word_bag = Counter(text.lower().split(" "))

             return word_bag

         from sklearn.model_selection import train_test_split

         data = load_data("train.txt")
         X, y = data.text, data.target
         X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)
         cls = SentimentClassifier(feature_method=bag_of_words, min_feature_ct=10)
         cls = cls.fit(X_train, y_train)
         print("Training set accuracy: ", cls.score(X_train, y_train))
         print("Dev set accuracy: ", cls.score(X_dev, y_dev))
```

```
Now training…
7000 6324


/usr/local/lib/python3.6/site-packages/sklearn/linear_model/_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


Training set accuracy:  0.9278571428571428
Dev set accuracy:  0.7643333333333333
```

```
In [10]: """
         Run this cell to test your model
         """
         from sklearn.model_selection import train_test_split

         data = load_data("train.txt")
         X, y = data.text, data.target
         X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)
         cls = SentimentClassifier(feature_method=better_featurize, min_feature_ct=10,L2_reg=1.)
         cls = cls.fit(X_train, y_train)
         print("Training set accuracy: ", cls.score(X_train, y_train))
         print("Dev set accuracy: ", cls.score(X_dev, y_dev))
```

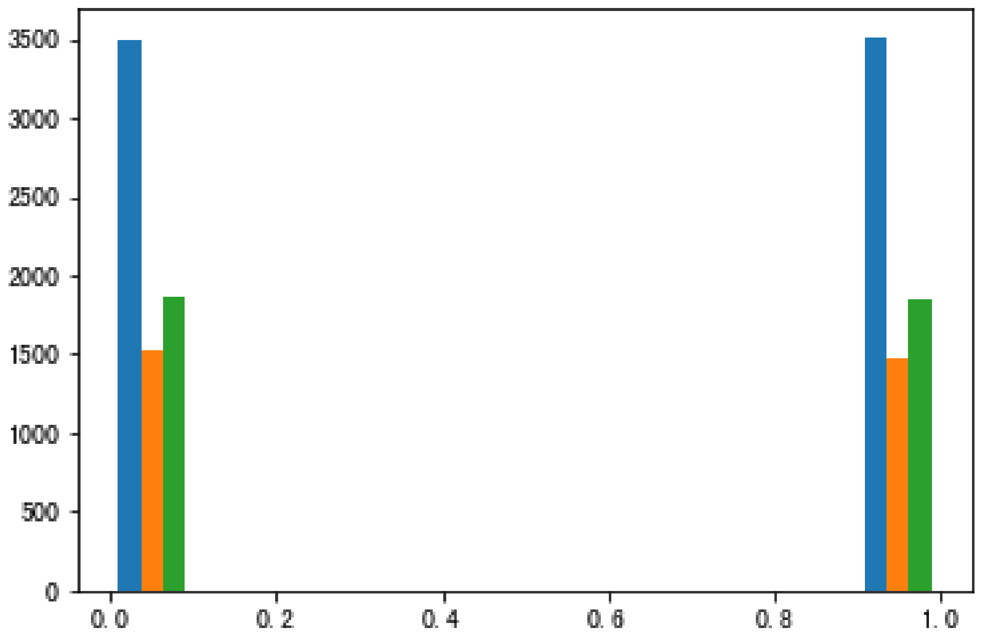
```
Now training…
7000 17344
Training set accuracy:  0.9071428571428571
Dev set accuracy:  0.793
```

```
In [18]: import matplotlib.pyplot as plt
         #y_pred = cls.predict(X_test)
         #plt.hist([y_train,y_dev,y_pred],label=['train', 'dev', 'test'])
         #plt.legend()
         bag_of_words_method = {'Logistic Regression': (0.928,0.764),"SVM":(0.994,0.738),"RandomF
         orestClassifier":(0.799,0.718),"XGBoost":(0.916,0.752)}
         own_method = {'Logistic Regression': (0.907,0.793),"SVM":(0.995,0.770),"RandomForestClas
         sifier":(0.791,0.724),"XGBoost":(0.955,0.732)}
         fig = plt.figure()
         ax = fig.add_axes([0,0,1,1])
         methods = bag_of_words_method.keys()
         train = [value[0] for value in bag_of_words_method.values()]
         dev = [value[1] for value in bag_of_words_method.values()]
         train1 = [value[0] for value in own_method.values()]
         dev1 = [value[1] for value in own_method.values()]
         X = np.arange(len(methods))*2
         ax.bar(X-0.375, train,width = 0.25)
         ax.bar(X-0.125, dev,width = 0.25)
         ax.bar(X+0.125, train1,width = 0.25)
         ax.bar(X+0.375, dev1,width = 0.25)
         ax.set_xticks(X)
         ax.set_xticklabels(methods)
         ax.legend(labels=['bow train', 'bow dev','own train','own dev'])
         plt.show()
```



```
In [16]: plt.hist([y_train, y_dev,cls.predict(X_test)]) # very balanced dataset! good
```

```
[16]: ([array([3485.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
               3515.]),
        array([1526.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
               1474.]),
        array([1861.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
               1853.])],
       array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
       <a list of 3 Lists of Patches objects>)
```



```
In [ ]:
```