task06-predict-hygiene

October 14, 2018

loganjtravis@gmail.com (Logan Travis)

1 Summary

From course page Week 5 > Task 6 Information > Task 6 Overview:

> In this task, you are going to predict whether a set of restaurants will pass the public health inspection tests given the corresponding Yelp text reviews along with some additional information such as the locations and cuisines offered in these restaurants. Making a prediction about an unobserved attribute using data mining techniques represents a wide range of important applications of data mining. Through working on this task, you will gain direct experience with such an application. Due to the flexibility of using as many indicators for prediction as possible, this would also give you an opportunity to potentially combine many different algorithms you have learned from the courses in the Data Mining Specialization to solve a real world problem and experiment with different methods to understand what's the most effective way of solving the problem.
>
> **About the Dataset** You should first download the dataset. The dataset is composed of a training subset containing 546 restaurants used for training your classifier, in addition to a testing subset of 12753 restaurants used for evaluating the performance of the classifier. In the training subset, you will be provided with a binary label for each restaurant, which indicates whether the restaurant has passed the latest public health inspection test or not, whereas for the testing subset, you will not have access to any labels. The dataset is spread across three files such that the first 546 lines in each file correspond to the training subset, and the rest are part of the testing subset. Below is a description of each file:
>
> - hygiene.dat: Each line contains the concatenated text reviews of one restaurant.
> - hygiene.dat.labels: For the first 546 lines, a binary label (0 or 1) is used where a 0 indicates that the restaurant has passed the latest public health inspection test, while a 1 means that the restaurant has failed the test. The rest of the lines have "[None]" in their label field implying that they are part of the testing subset.
> - hygiene.dat.additional: It is a CSV (Comma-Separated Values) file where the first value is a list containing the cuisines offered, the second value is the zip code, which gives an idea about the location, the third is the number of reviews, and the fourth is the average rating, which can vary between 0 and 5 (5 being the best).

2 Model 01: Logistic Regression of Unigram Probability

I start by representing text by creating a unigram form each restaurant's reviews then apply logistic regression. This simple model gives a useful baseline for future methods. It also highlights the difficulty of the prediction: Logistic regression alone proves an *incredibly* poor predictor!

2.1 Prepare Training Data

```
In [5]: # Split data into training and testing sets
        dfTrain["review_text_len"] = dfTrain.review_text.str.len()
        dfTrain, dfTest = train_test_split(dfTrain, test_size=0.3, random_state=84)
```

```
In [7]: # Sanity check on training versus testing split
        print("Training Data Statistics\n-----")
        print(dfTrain.groupby(["failed_hygiene"]).agg({
            "review_text": ["count"],
            "review_text_len": ["mean", "std"]
        }))
```

```
Training Data Statistics
-----
               review_text review_text_len
                     count            mean           std
failed_hygiene
False                  195     7276.015385  10327.798184
True                   187     9967.219251  12589.871449
```

```
In [8]: # Sanity check on training versus testing split
        print("Testing Data Statistics\n-----")
        print(dfTest.groupby(["failed_hygiene"]).agg({
            "review_text": ["count"],
            "review_text_len": ["mean", "std"]
        }))
```

```
Testing Data Statistics
-----
               review_text review_text_len
                     count            mean           std
failed_hygiene
False                   78     6230.256410   8406.959927
True                    86     9252.313953  10821.455737
```

2.2 Create Unigram Probability Matrix

I chose not to use IDF weighting. The training data concatenates all reviews for a restaurant with no delimiter to split them. Applying IDF instead of penalizing frequent terms across *documents* would penalize them across *restaurants*. This does not properly scale frequent terms.

Consider a term that appears once in every review versus a term that appears multiple times in one review for each restaurant. The term that appears in one review multiple times likely has greater predictive value than the term that appears in every review once. Unfortunately, penalizing by appearance in concatenated restaurant reviews would scale the two terms equally.

I instead count terms then normalize appearances within each restaurant creating a term probability matrix. Several additional comments:

- The SciKit Learn `CountVectorizer` does not lemmatize nor stem terms by default. I create my own tokenizer class to add those pre-processing steps.
- Logistic regression works best when the number of samples far exceeds the number of features. That is not the case for this data set. Expect poor performance with high sensitivity to model parameters.
- I did not tune the term vectorizer; it includes all terms found in the training data.
- I will tune the parameters in the next model. I intend this model as a *naïve* baseline.

```
In [9]: class MyTokenizer:
            def __init__(self):
                """String tokenizer utilizing lemmatizing and stemming."""
                self.wnl = nltk.stem.WordNetLemmatizer()

            def __call__(self, document):
                """Return tokens from a string."""
                return [self.wnl.lemmatize(token) for \
                                token in nltk.word_tokenize(document)]
```

```
In [10]: # Create TF vectorizer
         tf = CountVectorizer(max_df=1.0, min_df=1, \
                              stop_words="english", \
                              tokenizer=MyTokenizer())
```

```
In [11]: %%time

         # Calculate training term frequencies
         trainTerms = tf.fit_transform(dfTrain.review_text)

CPU times: user 7.78 s, sys: 266 ms, total: 8.05 s
Wall time: 8.21 s
```

```
In [12]: # Normalize for each restaurant
         trainP = trainTerms / trainTerms.sum(axis=1)
         print("{:,} restaurant reviews extracted into {:,} unigram terms.".format(*trainP.shape

382 restaurant reviews extracted into 23,107 unigram terms.
```

2.3 Train Logistic Regression Model

```
In [16]: %%time
```

```
         # Train logistic regression model
         model_TF_LR = model_TF_LR.fit(trainP, dfTrain.failed_hygiene)

CPU times: user 46.9 ms, sys: 15.6 ms, total: 62.5 ms
Wall time: 27.9 ms
```

```
In [18]: # Calculate F1 score
         model_TF_LR_Pred = model_TF_LR.predict(testP)
         printModelF1(dfTest.failed_hygiene, model_TF_LR_Pred, \
                      "Model 01: Logistic Regression of Unigram Probability")

Model 01: Logistic Regression of Unigram Probability
-----
F-1 Score: 0.000000
True Negatives: 77
True Positives: 0
False Negatives: 86
False Positives: 1
```

The simple logistic regression performed *horribly*. It only predict **one** failed hygiene inspection in the test data and that proved a false positive. I also tried other random seeds for the training/testing split. Some testing data sets yielded no predicted failed hygiene inspections.

Review text simply includes too much noise. While we anticipate hygiene issues to appear in reviews, we should also expect them to drown in a sea of non-hygiene related reviews about the food, service, location, etc.

3 Model 02: Recursive Feature Elimination Before Logistic Regression

I next try a feature selection method called Recursive Feature Elimination (RFE on SciKit Learn). The 22,000+ terms found in the training data far exceed the training sample (382). If a simple logistic regression has predictive value, it first needs to train on only the most useful features. Those features should also have strict independence.

RFE will not directly consider independence but it can *quickly* reduce the number of features to the most useful. I find the 30 top ranked features as a starting point to see what improvements logistic regression has to offer.

3.1 Find Most Predictive Terms using RFE

```
In [20]: # Create Recursive Feature Elimination instance
         rfe_TF_LR_RFE = RFE(model_TF_LR_RFE, n_features_to_select=30, step=100)

In [21]: %%time

         # Reduce features using Recursive Feature Elimination
         rfe_TF_LR_RFE = rfe_TF_LR_RFE.fit(trainP, dfTrain.failed_hygiene)

CPU times: user 1min 11s, sys: 16.4 s, total: 1min 28s
Wall time: 28.7 s
```

3.2 Train Logistic Regression after RFE

```
In [24]: %%time

         # Train logistic regression model
         model_TF_LR_RFE = model_TF_LR_RFE.fit(trainP_RFE, dfTrain.failed_hygiene)
```

```
CPU times: user 15.6 ms, sys: 0 ns, total: 15.6 ms
Wall time: 4.14 ms
```

```
In [25]: # Calculate F1 score
         model_TF_LR_RFE_Pred = model_TF_LR_RFE.predict(testP_RFE)
         printModelF1(dfTest.failed_hygiene, model_TF_LR_RFE_Pred, \
                      "Model 02: Recursive Feature Elimination Before Logistic Regression")
```

```
Model 02: Recursive Feature Elimination Before Logistic Regression
-----
F-1 Score: 0.442857
True Negatives: 55
True Positives: 31
False Negatives: 55
False Positives: 23
```

Restricting the logistic model to the 30 best terms improves its performance significantly. The resulting logistic model still performs about as well as a fair coin flip. A better technique to generate the most predictive features should improve model quality further.

4 Model 03: Latent Symantic Analysis of Unigram Frequency Before Logistic Regression

RFE selects the best features from an existing data set. The features it removes do not predict *as well* as the features it keeps but they can still have predictive value. Additionally, the remaining features may have significant covariance that would reduce the quality of logistic regression.

I therefore try a feature decomposition method called Latent Symantic Analysis (`TruncatedSVD` in SciKit Learn). The decomposition process still reduces the number of features but does so by linear combination of those features. Some ability to explain variation still gets loss. Usually much less than feature selection. The resulting decomposed features also have significantly less covariance.

I tuned the number of decomposed features to 180. LSA frequently starts with 100 but I found - through trial and error - 180 produced the best results.

Note: I perform LSA on the unigram frequency not probability within each restaurant's reviews. LSA on the unigram probabilities yielded worse results. Using frequencies does present a problem that I discuss in summarizing this model.

4.1 Perform LSA of Unigram Frequencies

```
In [26]: # Create Latent Semantic Analysis instance
         decomp_LSA = TruncatedSVD(n_components=180, random_state=42)

In [27]: %%time

         # Perform Latent Semantic Analysis on training terms
         decomp_LSA = decomp_LSA.fit(trainTerms)

CPU times: user 5.08 s, sys: 1.06 s, total: 6.14 s
Wall time: 1.88 s
```

4.2 Train Logistic Regression on LSA Features

```
In [31]: %%time

         # Train logistic regression model on LSA features
         model_LSA_LR = model_LSA_LR.fit(trainTermsLSA, dfTrain.failed_hygiene)

CPU times: user 46.9 ms, sys: 0 ns, total: 46.9 ms
Wall time: 47.6 ms


In [32]: # Calculate F1 score
         model_LSA_LR_Pred = model_LSA_LR.predict(testTermsLSA)
         printModelF1(dfTest.failed_hygiene, model_LSA_LR_Pred, \
                      "Model 03: Latent Symantic Analysis of Unigram Frequency Before Logistic R

Model 03: Latent Symantic Analysis of Unigram Frequency Before Logistic Regression
-----
F-1 Score: 0.674419
True Negatives: 50
True Positives: 58
False Negatives: 28
False Positives: 28
```

Logistic regression on the LSA features yields reasonable predictive value. I would recommend this model to a county hygiene inspector especially one with more restaurants to inspect than time. It has a higher than ideal false negative rate (predicting no hygiene issues when restaurant would fail its next inspection) **but** the model scales well. All the calculations can run in parallel after initial training even updating the unigram frequencies for a restaurant.

The model *may* require regular re-training. Term frequencies should increase over time as a restaurant receives more reviews. The logistic regression coefficients *may* therefore need adjustment to the higher term frequencies.

I emphasize "may" because the success of this model raised a question: Do most restaurants fail hygiene inspections earlier in their history? The training data does not include the date a restaurant opened so I cannot answer that question. However, we can hypothesize that the term frequency LSA into logistic regression model tips toward failed hygiene inspection on overall term frequencies as opposed to the frequencies of specific, more predictive terms. Diving into the LSA components and related terms exceeds the scope of this assignment (to build predictive models).

5 Model 04: Latent Symantic Analysis of Phrase Frequency Before Logistic Regression

Representing text as unigrams or even LSA features from those unigram vectors may not capture the best indicators. A phrase like "greasy glass" suggests bad hygiene more than "greasy" and "glass" do separately. I therefore generate a list of frequent phrases (up to trigrams) using AutoPhrase[1][2], an improved version of SegPhrase. I then feed the frequency of those phrases through a similar LSA and logistic regression to compare with the previous model.

Note: AutoPhrase uses Java which I cannot easily run inside this notebook. Please see below for my command line parameters and execution log.

```
MODEL='./models/hygiene' RAW_TRAIN='./wip/train_hygiene.dat' RAW_LABEL_FILE='./wip/train_hygiene
```

```
In [77]: # Print AutoPhrase log file
         with open(AUTOPHRASE_LOG, "r") as f:
             print(f.read())
```

```
===Compilation===
===Tokenization===
Current step: Tokenizing input file...
real        0m1.996s
user        0m5.688s
sys        0m0.797s
Detected Language: EN
Current step: Tokenizing stopword file...
Current step: Tokenizing wikipedia phrases...
Current step: Tokenizing expert labels...
com.cybozu.labs.langdetect.LangDetectException: no features in text
        at com.cybozu.labs.langdetect.Detector.detectBlock(Detector.java:235)
        at com.cybozu.labs.langdetect.Detector.getProbabilities(Detector.java:221)
        at com.cybozu.labs.langdetect.Detector.detect(Detector.java:209)
        at Tokenizer.detectLanguage(Tokenizer.java:151)
        at Tokenizer.main(Tokenizer.java:824)
Using default setting for unknown languages...
Using default setting for unknown languages...
Using default setting for unknown languages...
Using default setting for unknown languages...
Using default setting for unknown languages...
===Part-Of-Speech Tagging===
Current step: Splitting files...
Current step: Tagging...
Current step: Merging...
```

```
===AutoPhrasing===
=== Current Settings ===
Iterations = 2
Minimum Support Threshold = 10
Maximum Length Threshold = 6
POS-Tagging Mode Enabled
Number of threads = 10
Labeling Method = DPDN
        Auto labels from knowledge bases
        Max Positive Samples = -1
=======
Loading data...
# of total tokens = 1024854
max word token id = 27087
# of documents = 546
# of distinct POS tags = 57
Mining frequent phrases...
selected MAGIC = 27091
# of frequent phrases = 34509
Extracting features...
Constructing label pools...
        The size of the positive pool = 3409
        The size of the negative pool = 30845
# truth patterns = 49379
Estimating Phrase Quality...
Segmenting...
Rectifying features...
Estimating Phrase Quality...
Segmenting...
Dumping results...
Done.

real        0m8.071s
user        0m18.734s
sys        0m3.109s
===Saving Model and Results===
===Generating Output===
```

```
In [36]: # Create phrase frequency vectorizer
         pf = CountVectorizer(max_df=1.0, min_df=1, \
                              stop_words="english", \
                              tokenizer=MyTokenizer(), \
                              vocabulary=phrases)
```

```
In [37]: %%time
```

```
         # Calculate training phrase frequencies
         trainPhrases = pf.fit_transform(dfTrain.review_text)

CPU times: user 6.91 s, sys: 31.2 ms, total: 6.94 s
Wall time: 6.88 s
```

```
In [39]: # Create Latent Semantic Analysis instance
         decomp_LSAPhrases = TruncatedSVD(n_components=180, random_state=42)

In [40]: %%time

         # Perform Latent Semantic Analysis on training phrases
         decomp_LSAPhrases = decomp_LSAPhrases.fit(trainPhrases)

CPU times: user 891 ms, sys: 62.5 ms, total: 953 ms
Wall time: 297 ms
```

```
In [44]: %%time

         # Train logistic regression model on phrase LSA features
         model_Phrase_LSA_LR = model_LSA_LR.fit(trainPhrasesLSA, dfTrain.failed_hygiene)

CPU times: user 125 ms, sys: 0 ns, total: 125 ms
Wall time: 39.4 ms
```

```
In [45]: # Calculate F1 score
         model_Phrase_LSA_LR_Pred = model_Phrase_LSA_LR.predict(testPhrasesLSA)
         printModelF1(dfTest.failed_hygiene, model_Phrase_LSA_LR_Pred, \
                      "Model 04: Latent Symantic Analysis of Phrase Frequency Before Logistic Re

Model 04: Latent Symantic Analysis of Phrase Frequency Before Logistic Regression
-----
F-1 Score: 0.647059
True Negatives: 49
True Positives: 55
False Negatives: 31
False Positives: 29
```

Using frequent phrases (up to trigrams) instead of unigrams does not significantly alter the effectiveness of the model. Logistic regression on review text *alone* likely cannot improve further.

6 Model 05: Append Restaurant Features to LSA of Unigram Frequency

The training data includes additional features for each restaurant: categories, zip code, review count, and mean review score. I append the review count and mean review score to the previously generated LSA components from unigram frequency. Including those non-review features may improve logistic regression.

6.1 Get Additional Features for Restaurants

```
In [47]: # Merge additional columns into training and testing
         # data sets
         dfTrain = dfTrain.merge(dfRestAdd, how="left", left_index=True, right_index=True)
         dfTest = dfTest.merge(dfRestAdd, how="left", left_index=True, right_index=True)
```

```
In [49]: # Sanity check on training versus testing split
         print("Training Data Statistics\n-----")
         print(dfTrain.groupby(["failed_hygiene"]).agg({
             "review_text": ["count"],
             "review_text_len": ["mean", "std"],
             "review_count": ["mean", "std"],
             "review_score_mean": ["mean", "std"]
         }))
```

```
Training Data Statistics
-----
               review_text review_text_len               review_count  \
                     count            mean           std         mean
failed_hygiene
False                  195     7276.015385  10327.798184    10.651282
True                   187     9967.219251  12589.871449    14.620321

                          review_score_mean
                      std              mean       std
failed_hygiene
False           15.645935          3.598666  0.776270
True            17.182456          3.609714  0.751327
```

```
In [50]: # Sanity check on training versus testing split
         print("Testing Data Statistics\n-----")
         print(dfTest.groupby(["failed_hygiene"]).agg({
             "review_text": ["count"],
             "review_text_len": ["mean", "std"],
             "review_count": ["mean", "std"],
             "review_score_mean": ["mean", "std"]
         }))
```

```
Testing Data Statistics
-----
               review_text review_text_len               review_count  \
                     count            mean           std         mean
failed_hygiene
False                   78     6230.256410   8406.959927     9.115385
True                    86     9252.313953  10821.455737    13.337209

                          review_score_mean
```

```
                      std             mean       std
failed_hygiene
False           11.058265         3.714102  0.781775
True            14.886718         3.465266  0.803280
```

6.2 Train Logistic Regression on LSA Plus Restaurant Features

```
In [53]: %%time

         # Train logistic regression model on LSA and mean
         # restaurant score and number of reviews
         model_LSA_Plus_LR = model_LSA_Plus_LR.fit(\
                 trainTermsLSA_Plus, dfTrain.failed_hygiene)
```

```
CPU times: user 93.8 ms, sys: 0 ns, total: 93.8 ms
Wall time: 64.9 ms
```

```
In [54]: # Calculate F1 score
         model_LSA_Plus_LR_Pred = model_LSA_Plus_LR.predict(testTermsLSA_Plus)
         printModelF1(dfTest.failed_hygiene, model_LSA_Plus_LR_Pred, \
                      "Model 05: Append Restaurant Features to LSA of Unigram Frequency")
```

```
Model 05: Append Restaurant Features to LSA of Unigram Frequency
-----
F-1 Score: 0.674419
True Negatives: 50
True Positives: 58
False Negatives: 28
False Positives: 28
```

Adding the restaurant review count and mean score produces **identical** results to the unigram LSA into logistic regression model (#03). That result surprised me. I anticipated the review count and mean score would improve the predictive quality of logistic regression. Two explanations come to mind for breaking that expectation:

- Most reviews do not emphasize hygiene *even* in their scores. A separate analysis on the individual reviews might dis/prove that hypothesis. As noted previously, this training data concatenates all reviews and provides only mean score.
- The scores and reviews lack proximity (in time) to failed hygiene inspections. A restaurant might have failed its hygiene inspection only days before compiling the training data just as easily as another restaurant failed an inspection years prior. Their concatenated reviews and mean scores would not capture the timing discrepancy.

7 Test Additional Algorithms

The previous models all employed logistic regression on different representation of review text and restaurant features. Model #03 using LSA on unigram frequencies proved the most predictive with an F1 of 0.67. I do not anticipate significantly better performance from other algorithms; concatenated review text makes a weak a predictor of failed hygiene inspection. Yet SciKit Learn standardizes the training and prediction for many different algorithms including K-Nearest Neighbors, Random Forests, and Naïve Bayes. I test several below. None improve on model #03 enough to warrant their additional complexity and processing times.

7.1 Model 06: K-Nearest Neighbor Classifier of Unigram Probability

```
In [57]: # Calculate F1 score
         model_KNN_Pred = model_KNN.predict(testP)
         printModelF1(dfTest.failed_hygiene, model_KNN_Pred, \
                      "Model 06: K-Nearest Neighbor Classifier of Unigram Probability")
```

```
Model 06: K-Nearest Neighbor Classifier of Unigram Probability
-----
F-1 Score: 0.521212
True Negatives: 42
True Positives: 43
False Negatives: 43
False Positives: 36
```

7.2 Model 07: Random Forest Classifier of Unigram Frequency

```
In [60]: # Calculate F1 score
         model_RF_Pred = model_RF.predict(testTerms)
         printModelF1(dfTest.failed_hygiene, model_RF_Pred, \
                      "Model 07: Random Forest Classifier of Unigram Frequency")
```

```
Model 07: Random Forest Classifier of Unigram Frequency
-----
F-1 Score: 0.627219
True Negatives: 48
True Positives: 53
False Negatives: 33
False Positives: 30
```

7.3 Model 08: Naïve Bayes of Binary Unigram Indicator

```
In [66]: # Calculate F1 score
         model_NB_Pred = model_NB.predict(testBi)
         printModelF1(dfTest.failed_hygiene, model_NB_Pred, \
                      "Model 08: Naïve Bayes of Binary Unigram Indicator")
```

```
Model 08: Naïve Bayes of Binary Unigram Indicator
-----
F-1 Score: 0.477612
True Negatives: 62
True Positives: 32
False Negatives: 54
False Positives: 16
```

7.4 Model 09: Latent Symantic Analysis of Binary Unigram Indicator Before Naïve Bayes

```
In [73]: # Calculate F1 score
         model_NB_LSA_Pred = model_NB_LSA.predict(testBiLSA)
         printModelF1(dfTest.failed_hygiene, model_NB_LSA_Pred, \
                      "Model 09: Latent Symantic Analysis of Binary Unigram Indicator Before Naï
```

```
Model 09: Latent Symantic Analysis of Binary Unigram Indicator Before Naïve Bayes
-----
F-1 Score: 0.617284
True Negatives: 52
True Positives: 50
False Negatives: 36
False Positives: 26
```

8 Summary: Best Model 03

Predicting hygiene inspection failure using logistic regression of LSA features generated from review unigram frequency proved the best model. It not only yielded the highest F1 score (0.67) but one of the lowest computing overheads in both time and memore. As an added benefit, the steps can execute in parallel by restaurant review (after training the model). Model parameters:

- Unigram frequency calculated with SciKit Learn `CountVectorizer`
  - Applies custom tokenizer to lemmatize and stem tokens using NLTK `WordNetLemmatizer`
  - Counts all tokens (i.e., minimum document frequency of 1 and maximum of 100%)
  - Removes English stop words
- Latent Symantic Analysis of unigram frequency with SciKit Learn `TruncatedSVD`
  - Decomposed into 180 components
  - Default randomized algorithm due to Halko (2009)
- Logistic Regression of LSA features with SciKit Learn `LogisticRegression`
  - Default to L2 penalty
  - No class weighting (i.e., assume pass/fail hygiene inspection have equal probability)

9 AutoPhrase Publications

AutoPhrase arose form the contributions of two publications:

1. Jingbo Shang, Jialu Liu, Meng Jiang, Xiang Ren, Clare R Voss, Jiawei Han, "Automated Phrase Mining from Massive Text Corpora", accepted by IEEE Transactions on Knowledge and Data Engineering, Feb. 2018.
2. Jialu Liu, *Jingbo Shang*, Chi Wang, Xiang Ren and Jiawei Han, "Mining Quality Phrases from Massive Text Corpora", Proc. of 2015 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'15), Melbourne, Australia, May 2015. (* equally contributed, slides)