CIS530 FINAL REPORT

STUART WAGNER & MICHAEL WOODS

1 INTRODUCTION

Text classification on the basis of high or low information density has until recently [1] been ignored. Building off the past research, we seek to improve accuracy on classifying sentences with high information density.

Information density has several applications. Text summarization has relied partially on the use of KL divergence–the measurement of a difference between one distribution and another. However, this measure fails to account of information density. Information dense sentences generally better summarize information, as they more concisely describe information and ideas. Combined with a summarizer, density classification could provide a more powerful and precise text summarization.

Nenkova and Yang used a number of features in developing their model. We used many similar features, however as students new to NLP, we instead thought that using a variety of learners could provide better results. We therefore approached the problem as a machine learning problem that leveraged NLP as feature inputs. Could we learn which features were more important? Could we leverage a variety of models, using ensemble learning, to improve accuracy? The answer to those questions, generally speaking, is yes.

1 Nenkova & Yang "Detecting Information-Dense Texts in Multiple News Domains" 2014 http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8430/8622

2 METHODS AND RESOURCES

We decided to frame the problem of classifying a given article as either information dense or sparse as primarilty a machine learning problem. As with most problems that can be addressed with the methodologies of machine learning, having some knowledge of the domain–in this case text classification–can provide useful insights when selecting and extract features from raw data. With the approaches we learned in CIS530 regarding the analysis of a text and its structural componnts, along with our own intuitions and experimental insights, we outline the various features, tooling, and resources we employed for the project below.

2.1 Data Resources

MRC PSYCHOLINGUISTIC DATABASE
http://websites.psychology.uwa.edu.au/school/MRCDatabase/uwa_mrc.htm

The MRC Psycholinguistic Database is a machine readable dictionary containing a number of linguistic and psycholinguistic attributes and annotation for each word [2]. Specifically, we primarily made use of the 4,923 lexicon included in the MRC distribution.

LINGUISTIC INQUIRY AND WORD COUNT (LIWC)
http://www.liwc.net
The Linguistic Inquiry and Word Count (LIWC) is a text analysis software program designed to calculate the degree to "which people use different categories of words across a wide array of texts, including emails, speeches, poems, or transcribed daily speech" [3].

2.2 Tools and Software Packages

STANFORD CORENLP
http://nlp.stanford.edu/software/corenlp.shtml
A suite of text processing tools capable of part-of- speech (POS) tagging, named entity recognition (NER), tokenization, lemmatization, sentiment analysis, as well as dependency and syntactic parse extraction.

SCIKIT-LEARN
scikit-learnhttp://scikit-learn.org/
A popular Python machine learning framework with implementations for a large selection of popular classifiers.

NUMPY
http://www.numpy.org
A scientific computing package for Python providing native n-dimensional array datatype, as well as various linear algebra, statistical, and general mathematical functions suitable for ""number crunching".

NLTK
http://www.nltk.org/

2 http://www.psych.rl.ac.uk/
3 http://www.liwc.net/

A popular Python library for working with natural language that includes a number of convenient text processing tools and built in corpora for multiple languages.

2.3 Features

Building off the work of Nenkova and Yang, we leveraged a number of similar features in our approach to the problem of textual information density classification.

(1) CORENLP WORD COUNT–BINARY BAG-OF-WORDS
9799 dimensions. All unique tokens occurring between `<word></word>` tags in the generated XML obtained from running all training and test files through the CoreNLP tool were collected into a list and converted to lowercase. Any tokens consisting of stop words, as defined by NLTK's English stop word list or punctuation characters were removed. The list was then sorted alphabetically in ascending order and each word was assigned an index. To construct the feature vector for a given lead, the text of the lead was processed in a manner similar to that of the XML output and a vector the size of the bag-of-words model was initialized with zeros. If the lead contained the i-th word in the bag-of-words model, the i-th position in the feature vector was set to 1.

(2) CORENLP SENTENCE INFORMATION
7 dimensions. To construct the feature vector, each processed lead had a number of pieces of information extracted from it directly or from the XML file generated from it using CoreNLP. These quantities define the features indicated below along with our prediction on it's impact on information density ($+/-$):

- The number of sentences per lead. ($-$)
- The number of non-unique tokens per lead. ($-$)
- The number of named entities recognized in the lead. ($+$)
- The number of nouns in the lead (tokens tagged as `NN`, `NNS`, `NNP`, or `NNPS`) ($+$)
- The number of words with over six letters. ($+$)
- The number of quotation marks appearing in the lead. ($+$)
- The aggregate absolute sentiment score of the lead. Each sentence was assigned a numeric score according to the following rubric: "very negative": 2, "negative": 1, "neutral": 0, "positive": 1, and "very positive": 2. The scores were then summed and the absolute value was taken. ($-$)

(3) LIWC
78 dimensions. Our hypothesis was that words indicating sentiment and emotions might indicate sentences that were less information dense. We thought these might be positively correlated with information density: present tense, numbers, impersonal pronouns, and causation. We also believed multiple attributes may be negatively correlated, including: 1st person usage, auxillary verbs and past tense, quantifiers, and positive and negative emotion.

(4) CORENLP PRODUCTION RULE COUNT–BINARY BAG-OF-WORDS
5654 dimensions. The syntactic parse generated by CoreNLP for each sentence from a given lead is extracted from the corresponding `<parse></parse>` tag in the corresponding XML file. A parse tree is constructed in the same manner as described in Homework 3, where only non- terminal nodes are considered. To construct the master list of production rules, the outlined process is repeated for every lead file in the training and test set and every unique production rule is collected into a list. Each rule is then assigned an index, 0 to $N-1$, where N is the number of unique production rules. Then, to construct a feature vector for a given lead, a 1xN vector of zeros is created and every production rule occurring in the sentences of the lead is extracted. For every rule in the master list, if the i-th rule occurs in the rules extracted from the lead, the i-th index in the feature vector is set to 1. The belief was that perhaps certain production rules may be related with information density.

(5) MRC DICTIONARY–BINARY BAG-OF-WORDS
4923 dimensions. The MRC Psycholinguistic database word dictionary consisting of 4,923 words was used to construct a binary bag-of-words model in a similar manner to the CoreNLP binary bag-of-words model. We also attempted counts normalized by length of lead, however there did not appear to be a significant difference in accuracy. The text of each lead was tokenized, converted to lowercase and stripped of stop words. The benefit of using this approach is that the dictionary is independent of the task and training data (Nenkova and Yang).

(6) DEPENDENCY RELATIONS–BINARY BAG-OF-WORDS
5000 dimensions. For each lead file in the training set, the <dep></dep> nodes were extracted from the corresponding CoreNLP XML files and converted to a 3-tuple consisting of the dependency relation type, the governor word, and the dependent word. Each tuple instance was counted, and the top 5000 dependency tuples were used to build a bag-of-words vector in much the same manner as features (1), (4) and (5). Constructing the feature vector for a given lead proceeded in much the same manner as (5) as well.

2.4 Models

Regarding the actual task of classification, we constructed several classification models using the features described above. The model configurations are as follows:

- Logistic regression with a L2 penalty and class weights 1: 0.58 and -1: 0.42 set to reflect the true distribution of labels in the test set.
- Linear SVM regression with L2 loss and penalty functions and class weights 1: 0.58 and -1: 0.42 set to reflect the true distribution of labels in the test set.
- Bernoulli naive bayes with default parameters.
- Ensemble method with the three classifiers described above. Each classifier makes a prediction for a given observation and a majority vote is taken to determine the final predicted label.

3 FINAL SYSTEM

3.1 Team

We are team "W^2", or as it appears on the leaderboard in ASCII, `W^2` in.

3.2 System Design

From the very onset of the project, we exercised great discipline in producing what we believe to a clean, extensible system for making predictions about text. The software could easily have been treated as disposable–and designed as such –since the predictions produced by the system are ultimately the real value and not the software itself. Our rationale for putting a large, up-front effort into properly designing a data pipeline was to introduce a lower mental overhead cost when quickly iterating various feature-sets and models as the project progressed. The effort paid of many times over, as creating new models and features was the same as defining a new Python module. The model or feature could then later be referred to by name and maniupulated through set of generic functions.

3.3 Models

All models are implemented as regular Python modules in the `project.models` package in the project `code` directory. Additionally, all model modules are expected to define the functions described below, as this provides a common interface to interact with the model implementation:

```
def preprocess(model_name, train_files, test_files)
def train(model_name, train_files,  labels, *args, **kwargs)
def predict(model_name, model_obj, test_files, *args, **kwargs)
```

3.3.1 *Lookup*

A module can then easily be looked up by name like so:

```
linear_model = get_model('linear_model')
```

which is defined in the following simple function which makes use of the `importlib` from the Python standard libary:

```
def get_model(name):
    return importlib.import_module('project.models.'+name)
```

3.4 Features

Much like models, features are also implemented as Python modules, as this allows for quick prototyping and iterative refirement of new feature types. In a similar manner to models described above, features (or more accurately feature-izers) can easily be resolved with the following function:

```
def get_feature(name): return
    importlib.import_module('project.features.'+name)
```

Features can also expose the following functions

```
def build(feature_name, *args, **kwargs)
def featurize(feature_name, feature_obj=None, *args, **kwargs)
```

where `build` performs the initial setup work needed to featurize an input, and `featurize` performs the actual conversion of an input into a feature vector.

3.5 Pipeline

The conceptual classification pipeline is outlined below in pseudocode, which describes the general flow of information in the system, initially from raw input text to final predicted labels:

```
TRAIN_FILES = framework.get_train_files()

if in-submission-mode:
    TEST_FILES = framework.get_test_files()
else:
    TEST_FILES = TRAIN_FILES

# if the model defines an optional preprocess() function,
# call it:
if is_preprocess_defined('foo')
    optional = preprocess('foo', TRAIN_FILES, TEST_FILES)
else:
    optional = []

model_obj = train('foo', TRAIN_FILES, TRAIN_LABELS, *optional)

return predict('foo', model_obj, TEST_FILES, *optional)
```

3.6 Testing

Finally, the above pipeline is invoked from within the context of a testing framework we designed. The framework itself provides the input files to the prediction pipeline, which may vary depending on whether an actual prediction submission is being generated. If a leaderboard submission is to be generated, then actual test data is supplied to the pipeline by the framework, otherwise some holdout from the training data is used as test data. After the predictions have been made, the framework provides basic reporting information regarding the accuracy, precision, and F-score of the results.

4 OUTCOMES

All experiments were conducted using the `scikit-learn` `cross_validation.train_test_split()`, function with 10% of the training data held out as the test set.

The all submission configurations and related results are summarized in the table below. All feature sets are referenced by number as given in section 2.3.

4.1 Submission results

Attempt	Accuracy	Model	Feature sets
1	75.6%	Logistic regression	1, 2, 3
2	76.0%	Logistic regression /w PCA(n=500)+gridsearch	2, 3, 4, 5
3	78.4%	Ensemble classifier (LR+SVM+NB)	2, 3, 4, 5
4	78.4%	Ensemble classifier (LR+SVM+NB)	1, 2, 3, 4, 5
5	79.6%	Ensemble classifier (LR+SVM+NB)	2, 3, 4, 5, 6

5 DISCUSSION OF RESULTS AND CONCLUSION

Overall, we are quite pleased with our results. Each test submission either resulted in a higher accuracy, or at the very least the same score. We believe this is due to the design of our system which allowed for quick a test–prototype–report cycle of various model and feature configurations