Natural Language Processing

Signal Data Science

In this lesson, we will focus on natural language processing (NLP), *i.e.*, the application of data science to human-generated text. To illustrate the diversity and depth of modern NLP, we will proceed through a series of classical exercises in both Python and R.

Natural language processing is particularly enjoyable because its results tend to be very human-interpretable. Be sure to note any project ideas which occur to you while working through this assignment.

Email spam classification

In 2009, Symantec estimated that almost 90% of global email traffic consisted entirely of spam.[1] Modern email providers make extensive use of machine learning techniques to automatically classify and divert spam emails from your inbox. Without those algorithms, the enormous amount of spam received on a daily basis would be overwhelming!

We will begin with naive Bayes spam filtering, one of the oldest methods of statistical spam classification. After writing our own naive Bayes classifier and implementing various improvements, we will then compare its performance to that of elastic net regularized logistic regression. For training our model, we will use the CSDMC2010 SPAM corpus, downloadable online and located in the `csdmc2010-spam` dataset folder.

- Download the CSDMC2010 SPAM dataset. Examine several emails labeled as spam and several emails labeled as ham. (We will refer to "not-spam" emails as "ham" for convenience.[2])

[1] Symantec, September 2009, Report #33, State of Spam: A Monthly Report: "Overall spam volumes averaged at 87 percent of all email messages in August 2009. Health spam decreased again this month and averaged at 6.73 percent, while over 29 percent of spam is Internet related spam. Holiday spam campaigns have begun leveraging Halloween and Christmas, following closely after Labor Day-related spam."

[2] "Ham" is a commonly used term for not-spam, not just something we made up.

- Load the CSDMC2010 SPAM training data into R, storing the text of each `.eml` file in a string. (You may find `list.files()` and `scan()` useful.) Use the entirety of each file, including the HTML tags and email headers.

Naive Bayes classification

Using a naive Bayes classifier for spam filtration dates as far back as 1998.[3] The basic idea behind Bayesian spam filtering to classify spam based on which words appear or don't appear in any given email. For example, words such as "Viagra" or "refinance" show up often in spam emails, whereas words such as "brunch" are more likely to show up in non-spam emails. Bayes' theorem can therefore be used (with some simplifying assumptions) to train a spam vs. not-spam classifier based on the presence or absence of various words.

Three important assumptions are made in Bayesian spam filtering:

1. First, each email is treated as a bag of words in which the *order* of words is irrelevant, which enormously simplifies the task at hand (otherwise we might want to look at every possible *pairing* of words, or every possible *triplet* of words, and so on and so forth).
2. Second, we consider only the *presence* or *absence* of each word. That is, if the word is present, we do not consider its *frequency* in the email.
3. Third, the presence of each word in an email is assumed to be *statistically independent* from the presence of each other word. The assumption of *independence of events* is the core assumption of the naive Bayes method, turning a very computationally difficult problem into a tractable one.

We will proceed to process the data accordingly.

- Randomly select 80% of the emails as a training set, leaving the remaining 20% as the test set.
- Create two data frames from the training and test sets such that each row corresponds to a single email, each column corresponds to a particular word, and each entry is 0 or 1 depending on whether or not the column's corresponding word is present in the row's corresponding email. For now, consider a "word" to be any sequence of non-space characters without further characters on either side; *e.g.*, in `"Lorem4 ipsum; dolor. Sit <b>amet</b>"`, the words are `"Lorem4"`, `"ipsum;"`, `"dolor."`, `"Sit"`, and `"<b>amet</b>"`. The columns for the two data frames should *not*

[3] See Sahami *et al.* (1998), A Bayesian Approach to Filtering Junk E-Mail: "In addressing the growing problem of junk E-mail onthe Internet, we examine methods for the automated construction of filters to eliminate such unwanted messages from a user's mail stream. By casting this problem in a decision theoretic framework, we are able to make use of probabilistic learning methods in conjunction with a notion of differential misclassification cost to produce filters which are especially appropriate for the nuances of this task."

be identical (*i.e.*, there should be words in the training set which do not appear in the test set and vice versa). (You may find `strsplit()` helpful.)

Consider a subset of the words in the entire training corpus consisting of n words $w_1, w_2, \ldots, w_n$. Let W_i, S, and H respectively denote events corresponding to w_i being present in an email, an email being spam, and an email being ham, and let W denote the *conjunction* of all the events W_i, *i.e.*, the event corresponding to $w_1, w_2, \ldots, w_n$ all being present in an email. From Bayes' theorem, we can write an expression for $P(W \mid S) = P(W_1, W_2, \ldots, W_n \mid S)$, the probability of observing the words $w_1, w_2, \ldots, w_n$ in a spam email:

$$P(W \mid S) = \frac{P(S \mid W)P(W)}{P(W)} = \frac{P(W \mid S)P(S)}{P(W \mid S)P(S) + P(W \mid H)P(H)}.$$

It is simplest to assume *a priori* that any given email is equally as likely to be spam or ham, *i.e.*, $P(S) = P(H) = \frac{1}{2}$. The above expression then simplifies to

$$P(S \mid W) = \frac{P(W \mid S)}{P(W \mid S) + P(W \mid H)}.$$

Although we only have to calculate the two probabilities $P(W \mid S)$ and $P(W \mid H)$, they are both computationally intractable. First, each of the two would have to be evaluated for every possible combination of words W from the entire corpus, which correpsonds to an enormously large number of parameters to estimate; second, the amount of data required to obtain accurate estimations of such probabilities would be enormous. (Intuitively, if we pick 100 random words from the dictionary, a very small proportion of emails *in general* will contain those 100 words specifically! As such, we would need a *lot* of emails to be able to estimate $P(W \mid S)$ and $P(W \mid H)$.)

However, we can make a simplifying assumption to turn our *full Bayes* classification problem into a *naive Bayes* classification problem. Specifically, we assume that the presence of each word is *independent* of the presence of each other word, so

$$P(W_1, W_2, \ldots, W_n \mid S) = P(W_1 \mid S)P(W_2 \mid S) \cdots P(W_n \mid S)$$

and similarly for ham emails as well, where $P(W_i \mid S)$ is simply the proportion of spam emails in the training set which contain the word w_i!

- For each word in the training set, calculate $P(S \mid)$

Suppose, for instance, that the words "birthday" and "celebration" are both present in ham emails with probabilities p_b and p_c. If the occurrences of "birthday" and "celebration" were *independent* of each other in ham emails—that is, knowing that one word was present in a ham email provided zero information

about how likely the other one was to be present—then an email consisting of both words would have

$$P(W \mid H) = P(W_\text{b} \text{ and } W_\text{c} \mid H) = P(W_\text{b} \mid H)P(W_\text{c} \mid H) = p_\text{b}p_\text{c}.$$

However, that assumption is *not* true; a ham email containing "birthday" is much more likely than usual to also contain "celebration" and vice versa. As such, multiplying together $

- For each word in the training set, calculate $P(S \mid W_i)$. If a word appears only in spam emails, assign it a probability of $P(S \mid W_) = 0.99$ instead of 1; similarly, if a word appears only in ham emails, assign it a probability of 0.01 rather than 0.

Our eventual goal is to calculate $P(S \mid W)$ for each email. This is difficult in general, because the occurrences of words may be *correlated*. For example, "birthday", "cake", and "exegesis" are more likely to occur in ham emails; however, since "birthday" and "cake" are likely to occur together (compared to "birthday" and "exegesis" or "cake" and "exegesis"), the presence of both of them provides little additional information over the presence of either one of them alone. As such, even if $P(S \mid W_i)$ were identical for all three of those words, we ought to rate an email containing "birthday" and "cake" as more likely to be spam than one containing "birthday" and "exegesis" or "cake" and "exegesis".

In fact, determining $P(S \mid W_i)$ for each word alone is *not sufficient* for a full computation of $P(S)$; we really want to calculate $P(S \mid W_1, W_2)$ for every possible pair of words, $P(S \mid W_1, W_2, W_3)$ for every possible triple, and so on and so forth. Aside from being an enormous computational task, it would also require a tremendous amount of data to get an accurate estimate of $P(S \mid W_1, W_2, \ldots, W_n)$ for high values of n. (Intuitively, if we pick 10 random words from the dictionary, a very small proportion of emails *in general* will contain all 10 of those words!)

However, we can continue onwards and turn *full Bayes* classifier into a *naive Bayes* classifier by assuming that the occurrence of each word is *independent* of the occurrences of other words. This is almost surely false, but in practice this simplification works remarkably well! From this assumption, it follows that an email with words $w_1, w_2, \ldots, w_n$ has the associated probabilities

$$P(S) = \frac{p_1 p_2 \cdots p_n}{p_1 p_2 \cdots p_n + (1 - p_1)(1 - p_2) \cdots (1 - p_n)},$$

where $p_i = P(S \mid W_i)$, the probability that email e_p is spam given that it contains the word w_i and $w_1, w_2, \ldots, w_n$ are the words in e_p.

Due to floating-point underflow, instead of calculating $P(S)$ directly, it is better to calculate

$$\log\left(\frac{1}{P(S)} - 1\right) = \sum_{i=1}^{N} (\log(1 - p_i) - \log p_i)$$

because the summation doesn't have problems with underflow due to multiplying many small numbers together, and to then calculate $P(S)$ after a numeric expression for the right side of the above equation has been obtained.

- Look at the words with the highest and lowest p_is. Interpret the results.
- Calculate the true positive, true negative, false positive, and false negative rates for your classifier.
- Modify your classifier so that it converts all uppercase characters to lowercase. Does this improve the performance of your classifier on the training data?
- Find examples of both spam and non-spam emails from your personal email accounts. See if your classifer classifies them correctly.

Using n-grams with logistic regression

We can compare our naive Bayes classifier with logistic regression. For our logistic regression, we will use as features the frequency counts of individual words, *i.e.*, the number of time each word we know appears in an email. Additionally, we will also use n-grams, which are sequences of n consecutive words.

- Use the `ngram` package to create a dataframe of 1-grams and 2-grams from the training data with the `ngram()` and `get.phrasetable()` functions. Each row should represent a particular email and each column should be one of the 1-grams or 2-grams.
- Use regularized elastic net logistic regression to predict spam vs. not-spam, selecting the hyperparameters α and λ with the `caret` package.
  - To reduce computational demands, restrict consideration to the 1000 most common n-grams.
- Compute the true positive, true negative, false positive, and false negative rates for you logistic regression spam classifier. Compare its performance to that of your naive Bayes spam classifier.

Analysis of Github commit logs

We will use the Github API to scrape the commit logs for Linus Torvalds, creator of the Linux kernel, and Bram Moolenaar, creator, maintainer, and benevolent

dictator for life of the Vim text editor. Afterward, we will perform sentiment analysis on their commit messages.

Using the Github API

Since Linus and Bram make most of their commits to Linux and Vim respectively, we can use the API to (1) get some of the latest commits to Linux and Vim and (2) strip out all of the commits which don't come from either of them.

The Github API can be accessed directly via your browser. In general, you begin with the url `https://api.github.com/` and then successively append text to it, *e.g.*, `https://api.github.com/users/JonahSinick`.

- Referencing the API documentation on commits, figure out which API queries will return the latest commits for Linux and for Vim. (A parameter beginning with a colon (`:`) is a *variable* which you should fill in with the appropriate value.)
- Write a Python script to access the Github API and download our desired data. Follow these specifications:
  - Use `urllib.request` to download the results of API calls. Use the `json` module, particularly the `loads()` function, to strip out all the commit messages which don't come from Linus or Brad.
  - Write the commit messages to two files, `linus.txt` and `brad.txt`, with one message per line.

Performing sentiment analysis

- Load the files containing the commit messages for Linus and Brad. Process them, creating one vector for Linus's messages and another vector for Brad's messages.
- Install and load the `qdap` package, which has functions for both cleaning text and performing sentiment analysis.
- Following the Cleaning Text & Debugging vignette, use `qdap` to clean the commit messages in preparation for sentiment analysis. In particular, `check_text()` should suggest the usage of some text-cleaning functions to use.
- Combine all of the commit messages into a character vector with many entries. In addition, create a vector of labels (integers 0 or 1) which indicate whether the corresponding entry in the aforementioned character vector is from Linus or Brad.

- Use `polarity()` with its default settings to perform sentiment analysis, passing in both the character vector of every commit message as well as the grouping vector.

The results of the analysis are stored in `$all`, a data frame with a column `polarity` for the sentiment polarity score of each message.

- Plot two histograms of the polarity scores for Linus and Brad overlaid on top of each other. Interpret the results.
- Look at the commit messages which had the lowest and highest polarity scores.

Latent Dirichlet allocation on Wikipedia articles

The technique of Latent Dirichlet allocation (LDA) is analogous to performing factor analysis on text. Intuitively, it reads a large collection of documents – a *corpus* – and tries to find what the "topics" of the documents are.

In Python, you will scrape every article on Wikipedia which falls into the machine learning category and process the text. Afterward, you will run LDA on the corpus of text using R.

Overview of LDA

Wikipedia gives a good intuitive explanation of LDA:

> In LDA, each document may be viewed as a mixture of various topics. [...]
>
> For example, an LDA model might have topics that can be classified as **CAT_related** and **DOG_related**. A topic has probabilities of generating various words, such as *milk*, *meow*, and *kitten*, which can be classified and interpreted by the viewer as "CAT_related". Naturally, the word cat itself will have high probability given this topic. The **DOG_related topic** likewise has probabilities of generating each word: *puppy*, *bark*, and *bone* might have high probability. Words without special relevance, such as the (see function word), will have roughly even probability between classes (or can be placed into a separate category). A topic is not strongly defined, neither semantically nor epistemologically. It is identified on the basis of supervised labeling and (manual) pruning on the basis of their likelihood of co-occurrence. A lexical word may occur in several topics with a different probability, however, with a different typical set of neighboring words in each topic.

Precisely, we first pick a number of topics k. Next, we posit a *generative model* according to the following:

1. Topics are probability distributions over the set of words used in all documents. That is, each topic is represented by the assignment of a number between 0 and 1 to each distinct word such that the sum of all those numbers is 1. For example, if we have a very simple corpus that only has the words "a" and "b", then a possible topic T would be represented by $T(\text{"a"}) = 0.3$ and $T(\text{"b"}) = 0.7$.
2. Similarly, each document is a probability distribution over the set of topics.
3. Each document has a set number of words. Each word is generated as follows: First, we randomly pick a *topic* based on the proportions of the topics associated with its document. Next, we look at the probabilities associated with that topic and accordingly randomly choose one of the dictionary words.

The algorithm optimizes the probabilities associated with topics, documents, and words so as to maximize the *likelihood* associated with the training data. For each document, the associated distribution of topics is assumed to have a Dirichlet prior, hence the name of latent *Dirichlet* allocation. (The *latent* comes from the generative model falling into the class of *latent variable models*.)

Scraping Wikipedia pages

We will first scrape all of the text which we need.

In the following, I break down the tasks which you will perform in Python into a natural series of functions and loops. You should strive to test each of the functions you write as you write them, ensuring that the output each one returns is what you expect.

- Write a Python script to find the URL to every Wikipedia page in the machine learning category. Follow these specifications:
  - Use `urllib.request` to write a function `download_page(url)` which downloads the HTML of the page at `url` and returns it.
  - Using `download_page()`, download the Wikipedia page Category:Machine_learning. Write a function `get_urls(html)` which takes in the HTML of a Wikipedia category page and returns a dictionary with two entries: `pages`, a list of URLs to articles listed in the category page, and `subcategories`, a list of URLs to subcategories listed in the category page. Use Beautiful Soup to parse raw HTML. You can test your function on Category:Machine_learning to verify that it works.
  - Using `get_html()`, get a list of the links on Category:Machine_learning, and then add to the list the links on the *subcategories* of Cate-

gory:Machine_learning, and then add to the list the links on the subcategories of the subcategories of Category:Machine_learning, and so on and so forth until there are no more subcategories to traverse. Watch out for infinite loops as well as pages which branch out into overly general categories; both can be dealt with manually.

  - Write the list of article URLs to a text file, `wp_ml_urls.txt`, with one URL per line.

- Write a Python script to download each of the Wikipedia articles in the machine learning category. Follow these specifications:
  - We need to create a folder in which we can store our downloaded HTML files. Using the `os` and `shutil` modules, check if a folder called `raw_text` exists and delete it, along with all of its contents, if it does, and then create a new folder called `raw_text`.
  - Copy and paste your `download_page()` function from before into your current script. Write a function `download_article(url)` which downloads the Wikipedia article at `url`, parses the HTML with Beautiful Soup, and returns the text of the article without any of the HTML tags. (You may find the `.get_text()` function, available for Beautiful Soup HTML objects, helpful.)
  - Read in the text file of URLs which you created earlier. Iterate over the URLs and call `download_page()` on each one. For each url `[..]/en/Page_name`, save the associated text to `raw_text/page_name.txt`; that is, take everything in the URL after the last forward slash (`/`), remove all non-alphanumeric characters, make the string completely lowercase, and use that as the file name for the article's text.

We want to remove stop words, like "the", "is", "at", and "on" from our documents. Since they show up very frequently, they will dominate the topics in LDA and our results will be useless.

Moreover, we want to group together similar words, like "apples" with "apple" or "abaci" with "abacus". We have two methods available to us: *stemming* and *lemmatization*.

Stemming will strip away everything aside from the *stem* of a word. Sometimes, the stem itself is a word, like with "cats" $\rightarrow$ "cat". However, this is not guaranteed, like with "argue", "argues", "arguing" $\rightarrow$ "argu". It is often more convenient to use lemmatization, which does not simply cut off the end of words but rather reduces different inflections into the same base form. With lemmatization, "argue", "argues", and "arguing" all map to "argue". Since the base forms are all regular English words, the results of natural processing with lemmatized words is easy to read and interpret.

To that end:

- Write a Python script to further process the text of each of the downloaded Wikipedia. Follow these specifications:
  - Write a function `process_text(text)` which reads in a string `text` and returns its processed form, where you've removed all punctuation, converted everything to lowercase, removed stop words, and lemmatized the remaining words. Use the `nltk` package to do so, referring as necessary to the documentation on downloading NLTK corpora (which includes a stop word corpus) and on `nltk.stem`. (You can use the WordNet Lemmatizer implemented in NLTK.)
  - Like earlier, check for the existence of a folder called `processed_text`. If it exists, delete it along with its contents, and then create it.
  - Use `glob` to get a list of every `.txt` file saved in the `raw_text` folder.
  - Iterate over the paths to the text files. For each one, open the file, run its contents through `process_text()`, and save the results into the `processed_text` folder with the same filename (*e.g.*, `raw_text/alphago.txt` → `processed_text/alphago.txt`).
  - Examine your processed text files to ensure that the output is what you expect it to be.

Running LDA on Wikipedia pages

We are finally (almost) ready to run latent Dirichlet allocation on our text files!

First, we need to load them into R with the `tm` package, designed for natural language processing tasks.

- Load the corpus of text into R by running

  ```
  corpus = VCorpus(DirSource(directory=d, encoding="UTF-8"))
  ```

  where `d` is a string containing the path to your `processed_text` directory.
- Create a document-term matrix from your corpus by running `DocumentTermMatrix(corpus)`.
- Install and load the `topicmodels` package, which has an implementation of LDA.
- Select some arbitrary number of topics to use below 50 and run LDA on your document-term matrix with `LDA(dtm, k=num_topics)`. (This can take a couple minutes.)
- The easiest way to look at the results of LDA is to see which words have the highest "loadings" on each topic. View the top `n` terms of each topic in your LDA model with `terms(lda_model, n)`. Interpret the results.

- Calling `topics(lda_model, n)` will show the top `n` topics for each document (in terms of their "loadings"). Pick a couple of article which you find interesting and look at the topics associated with those documents. Are the results surprising or expected?
- Try running LDA again with a significantly different number of topics (2x difference or greater). How different are the topics? Are they more or less coherent than before?

Finding the right number of topics is difficult and there are many different ways to do so. Each model is associated with a *log-likelihood*, so the easiest way to choose the number of topics is to do a grid search over k and choose the one associated with the highest log-likelihood. One can also consider the perplexity metric, which is essentially measuring the same thing as the log-likelihood. There is also the `ldatuning` package, which calculates four different metrics for the "quality" of a LDA model. Finally, one can use the harmonic mean method to select the optimal k. Finally, Wallace *et al.* (2009) gives some even more complex (but better) methods for evaluating the quality of topic models.

For simplicity, we'll stick to the log-likelihood.

- Choose a reasonable grid for a grid search over values of k, storing each LDA model as you go. At the end, look at their associated log-likelihoods with the `logLik()` function. Plot the log-likelihoods against the number of topics k. If you have the time, do a second, finer grid search to get a better estimate of the optimal value of k. Examine the best model you've found in comparison with the other, less optimal models, using both `terms()` and `topics()`.

In reality, what you *should* be doing is estimating the log-likelihood via cross-validation, but it would simply be far too laborious to do so now. Good to know for the future, though.

Finally, note that it often produces better results to use n-grams, discarding the bag-of-words assumption. Using n-grams is computationally challenging, but there are ways to detect *phrases* in a text preprocessing step.

- Read David Mimno's Using phrases in Mallet topic models.

Closing notes

LDA is most useful for learning structure for corpora which are too large for humans to immediately fully understand. It has many extensions, such as *correlated topic models* which allow for greater correlations between topics or *dynamic topic models* which track the evolution of topics over time, with a huge amount of the work in this field being done by David Blei.

Some interesting applications of LDA include:

- Mimno and McCallum (2007), Organizing the OCA: Learning Faceted Subjects from a Library of Digital Books
- Hu and Saul (2003), A Probabilistic Topic Model for Unsupervised Learning of Musical Key-Profiles
- Pritchard *et al.* (2000), Inference of Population Structure Using Multilocus Genotype Data, which was written before the development of LDA as it is now but proposes essentially the same generative model