


Branch: master ▾

logistic_regression / Sentiment analysis with Logistic Regression.ipynb

Find file

Copy path

 **IISourcell** Add files via upload

48057b5 on 7 Aug 2018

1 contributor

1329 lines (1328 sloc) 98.4 KB

Sentiment analysis with Logistic Regression

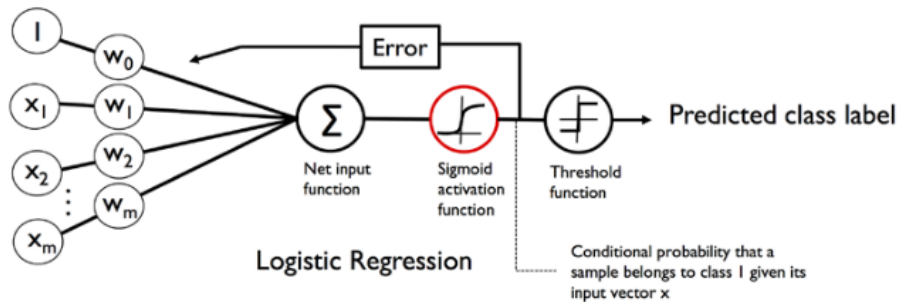
Logistic Regression

Some of the explanations in this tutorial are inspired by chapters 3 and 8 in the book *"Python Machine Learning"* by Sebastian Raschka

First, let's introduce what Logistic Regression is.

Logistic Regression (https://en.wikipedia.org/wiki/Logistic_regression) is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry too, which makes it attractive to play with.

Very simplistically explained, Logistic Regression works as follows:



First we will define the input for our algorithm. The input will be each sample in whatever dataset we are working with. Each sample will consist of several features. For example, if we're working with housing price prediction, the features for each sample could be the size of the house, number of rooms, etc. We'll call the input vector \mathbf{x} .

For the algorithm to learn, we need to define variables that we can adjust accordingly to what we want to predict. We will create a vector of *weights* (\mathbf{w}) that the model will adjust in order to predict more accurately. The process of adjusting those weights is what we call **learning**.

For every input sample, we will perform a dot product of the features by the weights \mathbf{xw} . This product is sometimes referred as *net input*. This will give us a real number. Since in this particular problem we want to *classify* (positive/negative), we need squash this number in the range $[0, 1]$. This will give us the *probability* of a positive event. A function that does precisely that is called **sigmoid**. The sigmoid function looks like this:

 sigmoid

What sigmoid is doing is basically transforming big inputs into a value close to 1, and small inputs into a value close to 0. This is exactly what we want.

We will do this for every sample in our training set and compute the errors. To calculate the error we only need to compare our prediction with the true label for each sample. We will sum the square errors of all the samples to get a global prediction error. This will be our **cost function**.

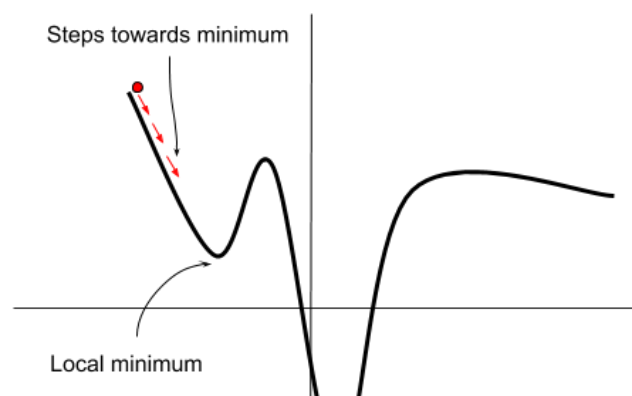
A cost function is then something we want to minimize. **Gradient descent** is a method for finding the minimum of a function of multiple variables, such like the one we're dealing with here.

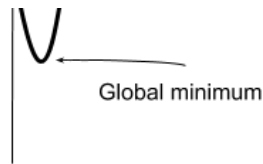
Gradient descent

Watch [this](https://www.youtube.com/watch?v=IHZwWFHwa-w) (<https://www.youtube.com/watch?v=IHZwWFHwa-w>) video for a visual introduction to Gradient Descent

Once all our training samples have been computed and the error calculated with our cost function, we need to *minimize* that cost function. A method for doing that is gradient descent. There are many [articles](https://en.wikipedia.org/wiki/Gradient_descent) (https://en.wikipedia.org/wiki/Gradient_descent) that contain detailed explanations and implementations of GD, so let's not do this here. However is good to have an intuition.

For illustration purposes, let's think about a function with two parameters. Something like this one:





Gradient descent will try to find the minimum of the function. To do so, we calculate the slope of the function at a certain point, and move towards the direction that makes the function decrease. There are some things to have in mind though.

As you can see a function can have one or several *local minimum*. In a local minimum, the slope will be zero and GD will "think" it's found the global minimum. To avoid this, we can choose a bigger "step" when we move towards the minimum. The "size" of the step towards the minimum is what we call the **learning rate**, and it's another adjustable parameter.

We need to be careful here: If we choose a too small learning rate, we can get stuck in a local minimum. If we instead choose a too big learning rate, we risk overshooting the global minimum. We need to experiment, and the adequate learning rate depends on the particular problem and the data.

The training process

In order for logistic regression to learn, we need to repeat the process described before several times. Each one of these times is called an **epoch**. The number of epochs to run depends on the problem and the training data. It is... yes, another tunnable parameter of the algorithm.

The set of all tunnable parameters is called **hyperparameters** of the model.

Like with the learning rate, we need to be careful when choosing the number of epochs: If we train too many epochs, we risk **overfitting**. This means that our model will "memorize" the training data and will generalize badly when presented new data.

If we train too little, it will fail to find any pattern and the prediction accuracy will be very low. This is known as **underfitting**.

There are techniques that help prevent overfitting. These **regularization** techniques are out of the scope of this tutorial, but... guess! It's also something to tune and experiment with :)

This is why when training a model you need to set aside a *test dataset* in order to know the accuracy of your algorithm in unknown data. The test dataset will **never** be used during training

Sentiment analysis

Let's first of all have a look at the data

```
In [1]: import pandas as pd

#Load the data into a DataFrame
train = pd.read_csv('data/train.csv', encoding='latin-1')
test = pd.read_csv('data/test.csv', encoding='latin-1')

train.head(10)
```

```
Out[1]:
```

| | ItemID | Sentiment | SentimentText |
|---|--------|-----------|---|
| 0 | 1 | 0 | is so sad for my APL frie... |
| 1 | 2 | 0 | I missed the New Moon trail... |
| 2 | 3 | 1 | omg its already 7:30 :O |
| 3 | 4 | 0 | .. Omgaga. Im sooo im gunna CRy. I'... |
| 4 | 5 | 0 | i think mi bf is cheating on me!!! ... |
| 5 | 6 | 0 | or i just worry too much? |
| 6 | 7 | 1 | Juuuuuuuuuuuuuuuuuuusssst Chillin!! |
| 7 | 8 | 0 | Sunny Again Work Tomorrow :- ... |
| 8 | 9 | 1 | handed in my uniform today . i miss you ... |
| 9 | 10 | 1 | hmmmm.... i wonder how she my number @-) |

As we can see, the structure of a tweet varies a lot between tweets. They have different lengths, letters, numbers, strange characters, etc.

It is also important to note that **a lot** of words are not correctly spelled, for example the word "Juuuuuuuuuuuuuuuuuuusssst" or the word "frie" instead of "friend"

This makes it hard to measure how positive or negative are the words within the corpus of tweets. If they were all correct dictionary words, we could use a **lexicon** to punctuate words. However because of the nature of social media language, we cannot do that.

So we need a way of scoring the words such that words that appear in positive tweets have greater score than those that appear in negative tweets.

But first... how do we represent the tweets as vectors we can input to our algorithm?

Bag of words

One thing we could do to represent the tweets as equal-sized vectors of numbers is the following:

- Create a list (vocabulary) with all the unique words in the whole corpus of tweets.
- We construct a feature vector from each tweet that contains the counts of how often each word occurs in the particular tweet

Note that since the unique words in each tweet represent only a small subset of all the words in the bag-of-words vocabulary, the feature vectors will mostly consist of zeros

Let's construct the bag of words. We will work with a smaller example for illustrative purposes, and at the end we will work with our real data.

```
In [2]: from sklearn.feature_extraction.text import CountVectorizer

twits = [
    'This is amazing!',
    'ML is the best, yes it is',
    'I am not sure about how this is going to end...'
]

count = CountVectorizer()
bag = count.fit_transform(twits)

count.vocabulary_
```

```
Out[2]: {'about': 0,
        'am': 1,
        'amazing': 2,
        'best': 3,
        'end': 4,
        'going': 5,
        'how': 6,
        'is': 7,
        'it': 8,
        'ml': 9,
        'not': 10,
        'sure': 11,
        'the': 12,
        'this': 13,
        'to': 14,
        'yes': 15}
```

As we can see from executing the preceding command, the vocabulary is stored in a Python dictionary that maps the unique words to integer indices. Next, let's print the feature vectors that we just created:

```
In [3]: bag.toarray()

Out[3]: array([[0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0],
               [0, 0, 0, 1, 0, 0, 0, 2, 1, 1, 0, 0, 1, 0, 0, 1],
               [1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0]], dtype=int64)
```

Each index position in the feature vectors corresponds to the integer values that are stored as dictionary items in the CountVectorizer vocabulary. For example, the first feature at index position 0 resembles the count of the word 'about', which only occurs in the last document, and the word 'is', at index position 7, occurs in all three tweets (two times in the second tweet). These values in the feature vectors are also called the **raw term frequencies**: $tf(t, d)$ —the number of times a term t occurs in a document d .

How relevant are words? Term frequency-inverse document frequency

We could use these raw term frequencies to score the words in our algorithm. There is a problem though: If a word is very frequent in *all* documents, then it probably doesn't carry a lot of information. In order to tackle this problem we can use **term frequency-inverse document frequency**, which will reduce the score the more frequent the word is across all tweets. It is calculated like this:

$$tf - idf(t, d) = tf(t, d) \cdot idf(t, d)$$

$tf(t, d)$ is the raw term frequency described above. $idf(t, d)$ is the inverse document frequency, then can be calculated as follows:

$$\log \frac{n_d}{1 + df(d, t)}$$

where n is the total number of documents and $df(t, d)$ is the number of documents where the term t appears.

The 1 addition in the denominator is just to avoid zero term for terms that appear in all documents. And the \log ensures that low frequency term don't get too much weight.

Fortunately for us scikit-learn does all those calculations for us:

```
In [4]: import numpy as np

        from sklearn.feature_extraction.text import TfidfTransformer
```

```
tfidf = TfidfTransformer(use_idf=True,
                          norm='l2',
                          smooth_idf=True)

np.set_printoptions(precision=2)

# Feed the tf-idf transformer with our previously created Bag of Words
tfidf.fit_transform(bag).toarray()
```

```
Out[4]: array([[ 0.    ,  0.    ,  0.72,  0.    ,  0.    ,  0.    ,  0.    ,  0.43,  0.    ,
                0.    ,  0.    ,  0.    ,  0.    ,  0.55,  0.    ,  0.    ],
               [ 0.    ,  0.    ,  0.    ,  0.4 ,  0.    ,  0.    ,  0.    ,  0.47,  0.4 ,
                0.4 ,  0.    ,  0.    ,  0.4 ,  0.    ,  0.    ,  0.4 ],
               [ 0.33,  0.33,  0.    ,  0.    ,  0.33,  0.33,  0.33,  0.2 ,  0.    ,
                0.    ,  0.33,  0.33,  0.    ,  0.25,  0.33,  0.    ]])
```

As you can see, words that appear in all documents like *is* (with 0.47), get a lower score than others that don't appear in all documents, like *amazing* (with 0.72).

Note also that `norm='l2'` parameter: This is an important one, and what is doing is normalize the tf-idfs so that they're all in the same scale and thus work better with Logistic Regression.

Data clean up (yay...)

Removing stop words

Now that we know how to format and score our input, we can start doing the analysis! Can we?... Well, we *can*, but let's look at our **real** vocabulary. Specifically, the most common words:

```
In [67]: from collections import Counter

vocab = Counter()
for twit in train.SentimentText:
    for word in twit.split(' '):
        vocab[word] += 1

vocab.most_common(20)
```

```
Out[67]: [(' ', 123916),
          ('I', 32879),
          ('to', 28810),
          ('the', 28087),
          ('a', 21321),
          ('you', 21180),
          ('i', 15995),
          ('and', 14565),
          ('it', 12818),
          ('my', 12385),
          ('for', 12149),
          ('in', 11199),
          ('is', 11185),
          ('of', 10326),
          ('that', 9181),
          ('on', 9020),
          ('have', 8991),
          ('me', 8255),
          ('so', 7612),
          ('but', 7220)]
```

As you can see, the most common words are meaningless in terms of sentiment: *I, to, the, and...* they don't give any information on positiveness or negativeness. They're basically **noise** that can most probably be eliminated. Let's see the whole distribution to convince ourselves of this:

```
In [56]: from bokeh.models import ColumnDataSource, LabelSet
from bokeh.plotting import figure, show, output_file
from bokeh.io import output_notebook
output_notebook()
```

(<https://bokeh.pydata.org>) Loading BokehJS ...

```
In [68]: import math

def plot_distribution(vocabulary):

    hist, edges = np.histogram(list(map(lambda x: math.log(x[1]), vocabulary.most_common())) , density=True, bins=500)

    p = figure(tools="pan,wheel_zoom,reset,save",
               toolbar_location="above",
               title="Word distribution across all twits")
    p.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:], line_color="#555555", )
    show(p)
```

```
plot_distribution(vocab)
```

It's clear now that a portion of the words are overly represented. These kind of words are called *stop words*, and it is a common practice to remove them when doing text analysis. Let's do it and see the distribution again:

```
In [60]: import nltk
```

```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data] /home/guillermo/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
```

```
Out[60]: True
```

```
In [70]: from nltk.corpus import stopwords
stop = stopwords.words('english')
```

```
vocab_reduced = Counter()
for w, c in vocab.items():
    if not w in stop:
        vocab_reduced[w]=c

vocab_reduced.most_common(20)
```

```
Out[70]: [(' ', 123916),
('I', 32879),
('I'm', 6416),
('like', 5086),
('-', 4922),
('get', 4864),
('u', 4194),
('good', 3953),
('love', 3494),
('know', 3472),
('go', 2990),
('see', 2868),
('one', 2787),
('got', 2774),
('think', 2613),
('&', 2556),
('lol', 2419),
('going', 2396),
('really', 2287),
('im', 2200)]
```

This looks better, only in the 20 most common words we already see words that make sense: *good, love, really*... Let's see the distribution now

```
In [72]: plot_distribution(vocab_reduced)
```

Removing special characters and "trash"

We still see a very uneven distribution. If you look closer, you'll see that we're also taking into consideration punctuation signs ('-', ',', etc) and other html tags like &. We can definitely remove them for the sentiment analysis, but we will try to keep the emoticons, since those *do* have a sentiment load:

```
In [80]: import re
```

```
def preprocessor(text):
    """ Return a cleaned version of text
    """
    # Remove HTML markup
    text = re.sub('<[^>]*>', '', text)
    # Save emoticons for later appending
    emoticons = re.findall('(?:[;|=])(?:-)?(?:\)|\(|D|P)', text)
    # Remove any non-word character and append the emoticons,
    # removing the nose character for standardization. Convert to lower case
    text = (re.sub('[\W]+', ' ', text.lower()) + ' '.join(emoticons).replace('-', ''))

    return text

print(preprocessor('This!! twit man :) is <b>nice</b>'))

this twit man is nice :)
```

We are almost ready! There is another trick we can use to reduce our vocabulary and consolidate words. If you think about it, words like: love, loving, etc. *Could* express the same positivity. If that was the case, we would be having two words in our vocabulary when we could have only one: lov. This process of reducing a word to its root is called **stemming**.

We also need a *tokenizer* to break down our tweets in individual words. We will implement two tokenizers: a regular one and one that does stemming.

we also need a *tokenizer* to break down our tweets in individual words. we will implement two tokenizers, a regular one and one that does stemming.

```
In [82]: from nltk.stem import PorterStemmer

porter = PorterStemmer()

def tokenizer(text):
    return text.split()

def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]

print(tokenizer('Hi there, I am loving this, like with a lot of love'))
print(tokenizer_porter('Hi there, I am loving this, like with a lot of love'))

['Hi', 'there,', 'I', 'am', 'loving', 'this,', 'like', 'with', 'a', 'lot', 'of', 'love']
['Hi', 'there,', 'I', 'am', 'love', 'this,', 'like', 'with', 'a', 'lot', 'of', 'love']
```

Training Logistic Regression

We are finally ready to train our algorithm. We need to choose the best hyperparameters like the *learning rate* or *regularization strength*. We also would like to know if our algorithm performs better stemming words or not, or removing html or not, etc...

To take these decisions methodically, we can use a Grid Search. Grid search is a method of training an algorithm with different variations of parameters to latter select the best combination

```
In [84]: from sklearn.model_selection import train_test_split

# split the dataset in train and test
X = train['SentimentText']
y = train['Sentiment']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0, stratify=y)
```

In the code line above, stratify will create a train set with the same class balance than the original set

```
In [85]: from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer(strip_accents=None,
                        lowercase=False,
                        preprocessor=None)

param_grid = [{ 'vect_ngram_range': [(1, 1)],
                  'vect_stop_words': [stop, None],
                  'vect_tokenizer': [tokenizer, tokenizer_porter],
                  'vect_preprocessor': [None, preprocessor],
                  'clf_penalty': ['l1', 'l2'],
                  'clf_C': [1.0, 10.0, 100.0]},
               { 'vect_ngram_range': [(1, 1)],
                  'vect_stop_words': [stop, None],
                  'vect_tokenizer': [tokenizer, tokenizer_porter],
                  'vect_preprocessor': [None, preprocessor],
                  'vect_use_idf': [False],
                  'vect_norm': [None],
                  'clf_penalty': ['l1', 'l2'],
                  'clf_C': [1.0, 10.0, 100.0]},
               ]

lr_tfidf = Pipeline([('vect', tfidf),
                    ('clf', LogisticRegression(random_state=0))])

gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=1,
                           n_jobs=-1)
```

```
In [86]: # Note: This may take a long while to execute, like... 1 or 2 hours
gs_lr_tfidf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 96 candidates, totalling 480 fits
[Parallel(n_jobs=-1)]: Done 42 tasks      | elapsed: 5.4min
[Parallel(n_jobs=-1)]: Done 192 tasks     | elapsed: 53.7min
[Parallel(n_jobs=-1)]: Done 442 tasks     | elapsed: 89.7min
[Parallel(n_jobs=-1)]: Done 480 out of 480 | elapsed: 97.9min finished
```

```
Out[86]: GridSearchCV(cv=5, error_score='raise',
                      estimator=Pipeline(memory=None,
                      steps=[('vect', TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                      dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                      lowercase=False, max_df=1.0, max_features=None, min_df=1,
```

```

        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=True,
        ...nalty='l2', random_state=0, solver='liblinear', tol=0.0001,
        verbose=0, warm_start=False)))
    fit_params=None, iid=True, n_jobs=-1,
    param_grid=[{'vect_ngram_range': [(1, 1)], 'vect_stop_words': [['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's...se_idf": [False], 'vect_norm': [None], 'clf_penalty': ['l1', 'l2'], 'clf__C': [1.0, 10.0, 100.0]}],
    pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
    scoring='accuracy', verbose=1)

```

```

In [87]: print('Best parameter set: ' + str(gs_lr_tfidf.best_params_))
        print('Best accuracy: %.3f' % gs_lr_tfidf.best_score_)

```

```

Best parameter set: {'clf__C': 1.0, 'clf_penalty': 'l2', 'vect_ngram_range': (1, 1), 'vect_preprocessor': <function preprocessor at 0x7f8daa8109d8>, 'vect_stop_words': None, 'vect_tokenizer': <function tokenizer at 0x7f8da0c3d378>}
Best accuracy: 0.772

```

Interestingly, the set of parameters that best results give us are:

- A regularization strength of 1.0 using l2 regularization
- Using our preprocessor (removing html, keeping emoticons, etc) *does* improve the performance
- Surprisingly, removing stop words does not improve accuracy
- word stemming doesn't seem to help either

As you can see, sometimes intuition may lead to wrong decisions, and it's important to *test* all our assumptions.

Let's see what's our best accuracy then:

```

In [88]: clf = gs_lr_tfidf.best_estimator_
        print('Accuracy in test: %.3f' % clf.score(X_test, y_test))

```

```

Accuracy in test: 0.770

```

If we would like to use the classifier in another place, or just not train it again and again everytime, we can save the model in a pickle file:

```

In [90]: import pickle
        import os

        pickle.dump(clf, open(os.path.join('data', 'logisticRegression.pkl'), 'wb'), protocol=4)

```

Finally, let's run some tests :-)

```

In [93]: twits = [
        "This is really bad, I don't like it at all",
        "I love this!",
        ":)",
        "I'm sad... :( "
    ]

    preds = clf.predict(twits)

    for i in range(len(twits)):
        print(f'{twits[i]} --> {preds[i]}')

```

```

This is really bad, I don't like it at all --> 0
I love this! --> 1
:) --> 1
I'm sad... :( --> 0

```