Published in Towards Data Science

Miguel Fernández Zafra Follow
Mar 23, 2020 · 9 min read · · Listen

Source

AN END TO END MACHINE LEARNING PROJECT

Deploying a Text Classification Model in Python

Learn to deploy a machine learning-based application with Dash and

Heroku

This article is the last of a series in which I cover **the whole process** of developing a machine learning project. If you have not read the previous two articles, I strongly encourage you to do it here and here.

The project involves the creation of a **real-time web application** that gathers data from several newspapers and shows a summary of the different topics that are being discussed in the news articles.

This is achieved with a supervised machine learning **classification model** that is able to predict the category of a given news article, a **web scraping method** that gets the latest news from the newspapers, and an interactive **web application** that shows the obtained results to the user.

As I explained in the first post of this series, the reason I'm writing these articles is because I've noticed that most of the times, the content published on the internet, books or literature regarding data science focus on the following: we have a labeled dataset and we train models to obtain a performance metric. This means crucial concepts such as data labeling or model deployment are **ignored**.

However, it turns out that machine learning and data science are intended to **solve problems** and **provide useful information**. So, having a model with 99% accuracy but not knowing how to take advantage of it will make us realize that we have **lost our time**.

That's why, in this series of posts, I try to cover, from the beginning to the very end, **all the necessary steps** to build a machine learning application from scratch that is useful to the final user and provides them with **valuable insights or information.**

So, the whole process of the development of this project has been divided into three different posts:

- Classification model training (link)
- News articles web scraping (link)

- App creation and deployment (this post)

The GitHub repo can be found here. It includes all the code and a complete report.

In the first article, we developed the **text classification model** in Python, which allowed us to get a certain news article text and predict its category with overall good accuracy.

In the second article, we created a script that **web-scraped** the latest news articles from different newspapers and stored the text.

In this post, we'll put all the pieces together and deploy our machine learning model so that it can provide **useful**, **handy** and **real-time insights** to the final user. We will follow these steps:

- Considerations before the deployment
- Creation of a Dash web application
- Deployment with Heroku
- Final thoughts

Considerations before the deployment

The data

At this point, we have trained a machine learning model with a dataset that contained news articles from 2004 to 2005. Our intention now is to apply this model to **live**, **current** data.

It is easy to understand that this may be the **first limitation** of our web application, and we'll see it with a simple example: suppose we run our web application and try to classify an article that talks about the new launch of the iPhone 11 and focuses on its amazing, state-of-the-art features such as a Retina display, 5G network support, etc...

Source

If we look back to 2004, one of the bestselling mobile phones was this one:

Source

So, it is probable that a lot of the terms that we find in the current data didn't even exist back in 2004 and, therefore, don't exist in our dataset.

As an example, this is an extract from an article in our sample:

> *Thousands of people queued for hours to get hold of one of the 200,000 PSPs which were shipped to retailers. The handheld console can play games, music and movies and goes on sale in Europe and North America next year. Despite the demand Sony said it would not increase the 500,000-strong stock of PSPs it plans to ship by year's end.*

Some of you won't even remember this portable console!

I hope this makes the following statement straightforward: **the more similar the data we used for training and the data we'll feed into the model once deployed are, the better**.

If the data on both stages is not similar, we will be addressing what is called a **data mismatch problem**. And, although I know this may sound too obvious, in practice I have seen a lot of models which didn't take into account this issue, which can lead to really bad performance with live data even though we had a really good one with training data and, consequently, to a **useless application**.

And, how can we solve this? Should we try to get even higher accuracy when training our model, by spending a lot of time creating features, or with hyperparameter tuning? You guessed it right: **No**. The only way to address this problem is to use a more updated corpus to train the model. And there's no magic machine learning can provide here.

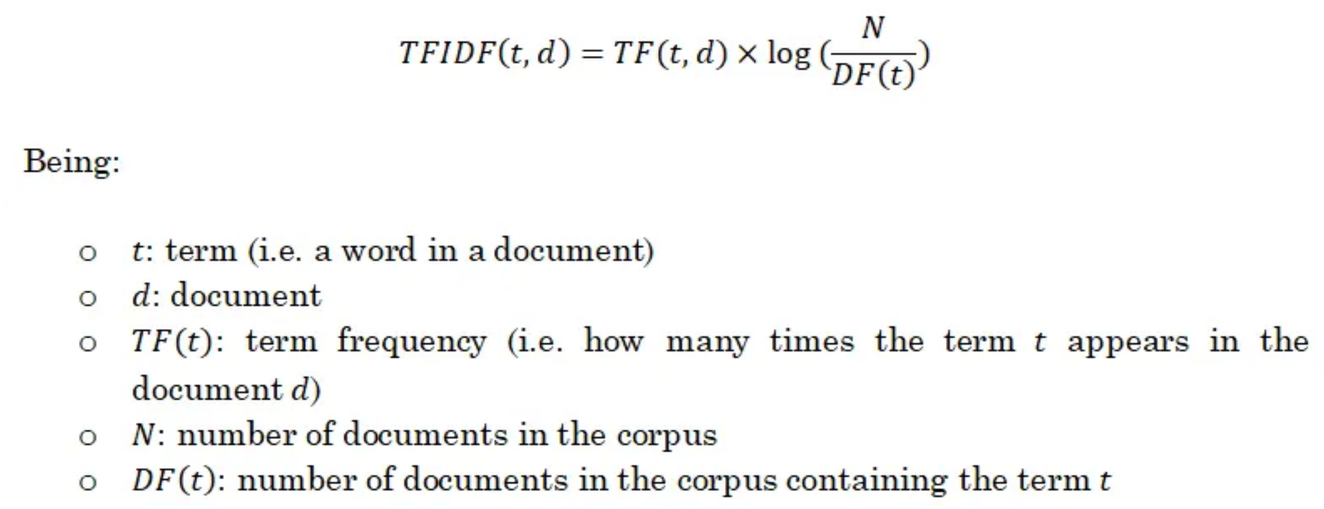
The features

In order to convert our raw text articles into numbers that could be fed into the machine learning model, we first cleaned the text in several steps: removing stop words, lemmatizing, etc… After that, we applied a *TF-IDF* vectorization to convert text to numeric features.

So, when we have our app deployed and get a new text from a newspaper, we will need to transform the raw text **exactly the same as we did**.

As an example, we used a pre-built stop word vector from NLTK. This means that when we run our web application, we will need to somehow have available this vector so that these words can be removed.

Regarding the *TF-IDF* vectorization, we have an important point here: as we saw in the first article, this vectorizer calculates, for a term *t* in a document *d,* the term frecuency in that document and the Inverse Document Frequency, which represents whether that term appears a lot in the corpus (and therefore it is a common term) or not (and it is an uncommon term so it is kind of "important").



$$TFIDF(t,d) = TF(t,d) \times \log\left(\frac{N}{DF(t)}\right)$$

Being:

- t: term (i.e. a word in a document)
- d: document
- $TF(t)$: term frequency (i.e. how many times the term t appears in the document d)
- N: number of documents in the corpus
- $DF(t)$: number of documents in the corpus containing the term t

Source: own creation

And it is important to note that this corpus we're referring to is the **training corpus**. This means that, when getting a single news article from the web-scraping process, the term frequency will be calculated only in that article, but the inverse document frequency will be calculated over the training corpus.

So, as it happened with the modern words associated with the newest iPhone, if we get a word from our live data that is not present in the training corpus, **we won't be able** to calculate a TF-IDF score. And it will be ignored. Again, we see the importance of the **data mismatch problem.**

The environment

We will cover it in a moment, but basically our web application will consist in a python script that will be executed and will give us with the results. So, as it happened with the NLTK stop words vector, we will need to create an environment where the script can be executed and has **all the necessary dependencies available:** from the basic libraries (sklearn, numpy, etc...) to the trained model, the TF-IDF vectorizer, etc…

The user experience

Finally, we need to take care of the user experience. So, as an example, if it takes us 2 minutes to scrape the latest 5 articles from a certain newspaper, we should probably not include it as an option for the user.

Once we have summarized all the things that we need to consider when deploying a model, it is clear that only focusing on getting a good accuracy in the training set and not taking into account all these considerations can lead us to a **useless web application**.

After discussing these issues, we'll see how to create the application with **Dash** and deploy it with **Heroku**. Dash and Heroku have really good documentation, so I won't spend too much time with the technical details here. Both can be learnt in a few hours by anyone, and I think the methodological considerations we have been covering are much more valuable.

Creation of a Dash web application

Dash is a productive Python framework for building web applications. It is written on top of Flask, Plotly.js and React.js, and it is pretty simple to use, but works amazingly. In the Dash webpage there is a really good tutorial that covers the creation of a web application from scratch explaining each and every step: the installation, the **layout** (what defines how our app looks like), the **callbacks** (which define the "things that happen") and some advanced concepts that will let us build the app just as we want it.

In order to create the code for the web application, we can execute it locally and the application will display in the browser. This way we can make easy changes and build the app as we want before deploying it.

The code that is under my web application can be found here.

After we have the code ready, the last step is to **deploy** the application.

Deployment with Heroku

There are many platforms that allow us to deploy a web application. Two good ones are Dash Enterprise and Heroku. I used Heroku since you can deploy an app for free (with some limitations), and for a small amount you can get a fully-operating server.

Again, Heroku has a really good tutorial that can be found here. It goes step by step covering all you need to deploy the app.

For this project, I deployed the app in Windows through anaconda. The steps I followed are:

```
# after signing in to Heroku and opening the anaconda prompt
# we create a new folder
$ mkdir dash-app-lnclass
$ cd dash-app-lnclass

# initialize the folder with git
$ git init
```

After that, we create an environment file (*environment.yml*) in which we will indicate the dependencies we are going to need:

```
name: dash_app_lnclass #Environment name
dependencies:
  - python=3.6
  - pip:
    - dash
    - dash-renderer
    - dash-core-components
    - dash-html-components
    - dash-table
    - plotly
    - gunicorn # for app deployment
    - nltk
    - scikit-learn
    - beautifulsoup4
    - requests
    - pandas
```

```
      - numpy
      - lxml
```

And activate the environment:

```
$ conda env create
$ activate dash_app_lnclass
```

Then, we initialize the folder with *app.py, requirements.txt* and a *Procfile:*

```
# the procfile must contain the following line of code
web: gunicorn app:server

# to create the requirements.txt file, we run the following:
$ pip freeze > requirements.txt
```

Finally, since we will be using nltk downloads (for the stopwords and other functions) and pickles, we need to add the *nltk.txt* file and the *Pickles* folder.

Finally we initialize Heroku, add the files to Git and deploy:

```
$ heroku create lnclass # change my-dash-app to a unique name
$ git add . # add all files to git
$ git commit -m 'Comment'
$ git push heroku master # deploy code to heroku
$ heroku ps:scale web=1  # run the app with a 1 heroku "dyno"
```

The detailed instructions can be found here.

Once we have followed all the steps, we will now have our web application **ready to use**! It can be found in this **link**.

Please take into account that this app has not been under periodic maintenance and therefore you may experience some errors.

Final thoughts

So this series of articles has come to an end. We have covered the **whole process** of creating a machine learning-based application, from getting the training data to creating the application and deploying it. I hope it has been useful and has illustrated all the steps and considerations that need to be taken into account when facing a new project. I have been exposing my thoughts and advice throughout the three articles, and they can be summarized in just a couple of paragraphs:

Don't ever lose sight of the objective you're pursuing: machine learning models are really fancy, but at the end, if they don't provide **some utility** they are **worthless**. So it is important to always keep in mind which is the utility we are providing to the user.

In addition, it is extremely beneficial to not lose the **end-to-end vision of the project**. As an example, when we are in the model training stage, we need to keep in mind things that are ahead that step, such as how is going to be the data when the app is deployed to avoid data mismatch.

An End To End Ml Project Python Text Classification Deployment

Hope these articles have been useful, and if you have any question don't hesitate to

Towards Data Science

240 1