Machine Learning Engineer Nanodegree
Capstone Project Report:
Automatic Tag Generation for StackOverflow Questions

by Pablo Campos Viana

I - Definition

a) Project Overview

Tag annotations have become a useful and powerful feature to facilitate item search in many websites and web applications. In many cases, the majority of tags assigned to items are supplied by users, a task which is time consuming and may result in annotations that are subjective and lack precision[1]. Therefore, finding good ways to perform automatic tagging is crucial in these scenarios and in recent years Machine Learning (ML) approaches have arisen.

Stack Overflow, according to its official site, *is the largest, most trusted online community for developers to learn, share their programming knowledge, and build their careers.* In the context of the Stack Overflow site, a tag is a word or phrase that describes the topic of the question and according to the help page about tagging of the site, they are a means of connecting experts with questions they will be able to answer by sorting questions into specific, well-defined categories, and they can be used to help users to identify questions that are interesting or relevant to them. This way, automated tagging is an important feature of the site since it improves user experience and fosters user engagement.

b) Problem Statement

In 2012, Stack Overflow released a competition to build ML models to predict which new questions asked on its site will be closed, along with data that can be used for other kind of tasks beyond the goal of the competition, for example: identifying tags from question text, predicting whether questions will be upvoted or downvoted based on its text, or predicting how long questions will take to answer. In this project I'm interested in the former one. In this context since each question can be associated with more than one tag, from a ML perspective, the problem can be approached as a multi-label classification one.

The problem to be solved can be stated as **automatic tag generation**, where, given text of questions, we want to predict their most probable tags.

---

c) Metrics

The evaluation metrics to be used are the standard classification metrics such as precision and recall, and their extensions to multi-label settings, this is, *precision@k* and *recall@k*, where *k* is the number of predictions requested for a given text query.

In this project, to simplify analysis, we will set a number of *k=3* predictions since we consider is a reasonable number of predicted tags to get for each question.

In fastText's built-in tool to compute metrics, *precision@k* and *recall@k* are computed sample-wise and then averaged. Following fastText's approach to compute the metrics, and since we will set *k=3* predicted tags, we will compute *precision@3* and *recall@k'* (where *k'* is the number of true tags for a particular sample) sample-wise and then get statistics over all the samples. More precisely, we will compute basic statistics (percentiles) instead of only one statistic (the average) to compare performance between models.

II - Analysis

Data Exploration & Visualization

We can perform a quick analysis of the data, specifically for the tags table. When we compute basic statistics over the frequencies over all the tags present in the table, we get the following:

	tag
count	3506.000000
mean	5.702510
std	27.213578
min	1.000000
25%	1.000000
50%	1.000000
75%	3.000000
max	862.000000

We observe that we have a total of 3506 different tags, an average of 5.7 occurrences, and more than half of tags have only one occurrence. In order to simplify analysis and modeling, we keep the top-100 tags. When we plot the histogram over the top-100 tags, we get the following:
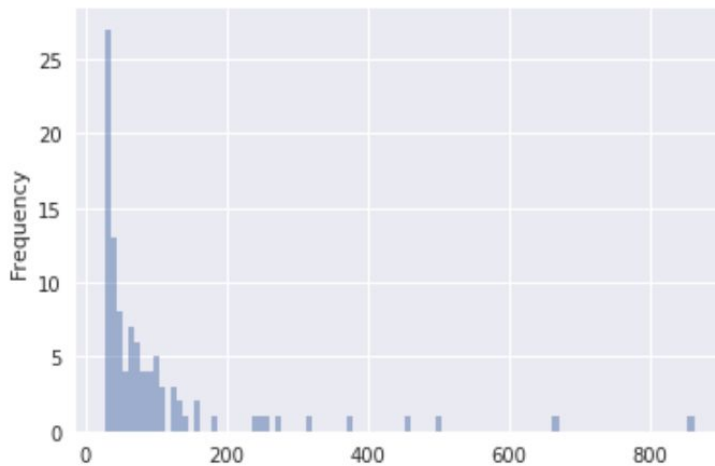
And we can observe that there are a few tags that have higher frequencies (> 200 occurrences) than the rest, which is expected since probably they correspond to the most popular programming languages. If we compute basic statistics as before but for the top-100 tags, we get the following:

	tag
count	100.000000
mean	99.230000
std	128.606506
min	29.000000
25%	37.000000
50%	56.000000
75%	98.250000
max	862.000000

And we can observe that now the average number of occurrences is around 99, and the minimum number of occurrences is 29. This way, we can be sure we will not have a strong problem regarding imbalance of classes when modeling.

Furthermore, we can draw a wordcloud over the top-100 tags to have an idea about the most frequent tags and their relative frequencies. The wordcloud is shown below.
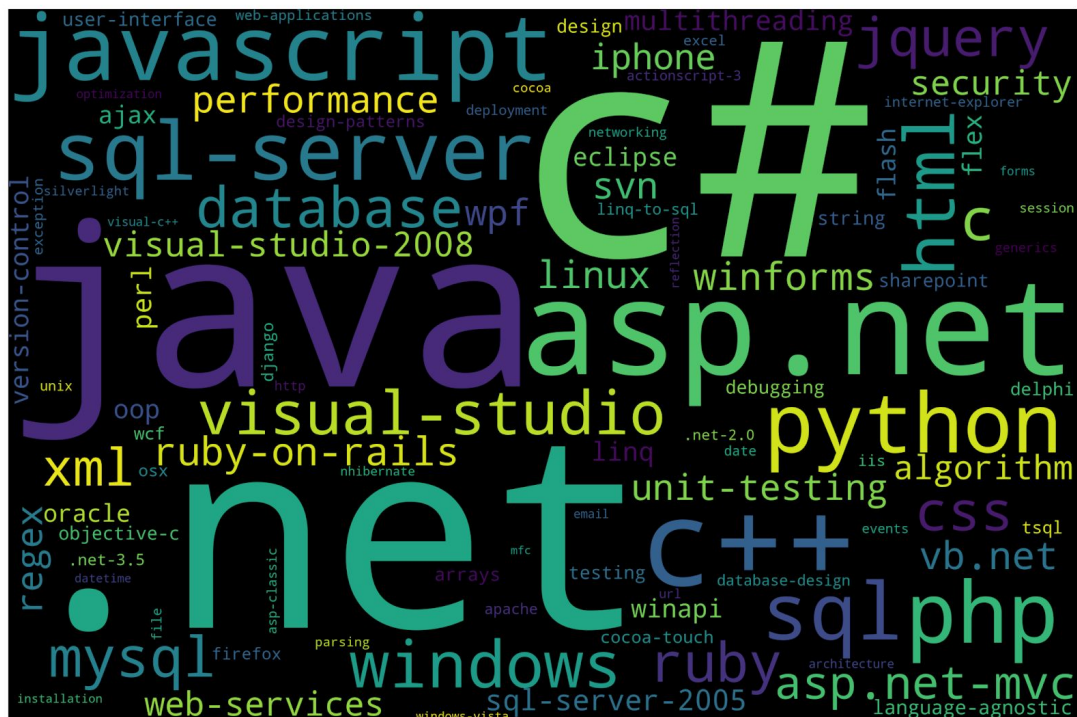
This way, we consider that using the top-100 tags is a good approach for this project. As possible and future improvements, we could investigate smart ways to deal with highly unbalanced classes, or to group the least frequent tags into one category, or to engineer a loss function to deal with many and unbalanced classes.

a) Algorithms and Techniques

The algorithms considered for this project, taking into account that the problem to solve is one of multi-label text classification, are Amazon's BlazingText and a LSTM network using PyTorch. The former is a built-in algorithm in Sagemaker, and since Sagemaker handles end-to-end training and deployment of custom PyTorch code, both of them can be easily trained and deployed and that is why we have chosen them for this project.

Although there are several approaches to deal with the multi-label problem such as those described in the Wikipedia article for multi-label classification, we have noted that fastText (and BlazingText) deal with the multi-label problem following a "pick one label at random" approach at training time.

In the case of the LSTM network, in order to simplify the choice and engineering of the network architecture and/or the loss function, and inspired by a fastText issue discussion, we had the idea to train the LSTM network like a traditional single-label classification one, using the same input text multiple times with their different labels each time at training time. We claim that this approach is similar to that of "pick one label at random" whenever the train/validation/test data splitting is performed in such a way that all observations for each user belong to the same split.

Additionally, for the PyTorch's LSTM network, we use the torchtext package, which consists of data processing utilities and popular datasets for natural language. In our case, this package helped us to build the vocabulary and perform padding at training time, and map the token strings to numerical identifiers and viceversa at test time.

b) Benchmark

An existing solution to the problem has been described by the developer Yi Ai in a Medium post. His solution uses fastText and we have noticed that is replicable. However, no hyperparameter tuning is performed and no metrics are reported. Nevertheless, with the settings described in his solution, we can replicate his model to use as baseline and evaluation metrics chosen for this project can be computed to use as benchmarks. At this moment, we have not found any other published solution for this particular problem and dataset.

III - Methodology

a) Data Preprocessing

The data acquisition and preprocessing steps performed can be summarized as follows:

- Download data using Kaggle's Public API via CLI, get questions and tags tables.
- Get titles text body from questions table, perform text cleaning and other standard NLP preprocessing.
- Match questions titles text (features) and tags text (labels) in one single table.
- Split table pseudorandomly into train, validation and test sets.
- Over these splits perform independently:
  a) Perform additional data preprocessing to leave the data splits with the format needed by both fastText and BlazingText, and store in a local directory.
  b) Perform additional data preprocessing to leave the data splits with the format needed by the PyTorch's LSTM network, and store in a local directory.
- Upload both directories to an Amazon S3 bucket.

b) Implementation

Benchmark: fastText

As described in the Benchmark subsection, an existing solution to the problem has been described by the developer Yi Ai in a Medium post. We use the same training settings reported in his post.
More precisely, we use the following parameters and values for training:

learning rate = 0.5
number of epochs = 25
length of n-grams = 2
embedding dimension = 100
loss = categorical cross-entropy

Algorithm 1: BlazingText

The training parameters of BlazingText are the same as fastText, since the former is based on the latter. For comparison reasons, we use the same values as described above for fastText, with the only difference that we allow BlazingText to use its early stopping criteria. However, when we performed training, there was not need to use the early stopping rule and the training procedure used all epochs.

Algorithm 2: PyTorch's LSTM network

We use a bidirectional LSTM network, based on a Medium post about a sentiment analysis task using PyTorch, and modify the architecture in order to allow score predictions for each label (tag) in the last layer.

The LSTM network has the following parameters and we describe the final values used for them:

input dimension = (vocabulary size)
embedding dimension = 100
output dimension = (number of different tags)
number of hidden layers = 2
hidden dimension = 32
dropout rate = 0.5
whether the network is bidirectional = False

With these parameters the network architecture is as follows:

Embedding layer   (input dimension x embedding dimension)
LSTM layer + Dropout (embedding dimension x hidden dimension)
LSTM layer + Dropout (embedding dimension x hidden dimension)
Fully connected   (hidden dimension x output dimension)

And for the training procedure we use the following setting:

optimizer = Adam (with learning rate=.001)
loss = categorical cross-entropy
number of epochs = 20

c) Refinement

Benchmark: fastText
Since the fastText model was used as baseline, it was used with the training settings as described in the previous subsection and no refinement was performed. As possible and future improvements, we could use fastText's automatic hyperparameter optimization tool to look for better configurations and a better model.

Algorithm 1: BlazingText
For a fair comparison, we used the same training settings as the fastText model as described in the previous subsection and no refinement was performed. As possible and future improvements, we could use Amazon SageMaker's Automatic Model Tuning tool to look for better configurations and a better model.

Algorithm 2: PyTorch's LSTM network
This model was the most challenging to develop since it implied to write custom training and prediction (serving) code, and implied many trial and error exercises about how to use properly torchtext package in conjunction with PyTorch. Once a working model was found, further improvements were made in two directions: 1) code refactoring and code modularization, and 2) better configurations of the parameters of the model. For the latter, although no formal tool was used to perform hyperparameter optimization, we performed several trial and error exercises where we wanted to balance good performance of the model, simplicity (e.g. we favor a not too deep model, with only 2 hidden layers, and a one directional LSTM layer was preferred instead of a bidirectional one), and training time.

IV - Results: Model Evaluation, Validation, and Justification

We can consider both quantitative and qualitative results and comparisons. The results are obtained over the same test set so we can be sure that they are comparable.

a) Quantitative Results

In the following table, the prefixes *ft*, *bt* and *pt* stand for "fastText", "BlazingText" and "PyTorch" respectively. The suffixes precision and recall stand for *precision@3* and *recall@k'* (where *k'* is the number of true tags for a particular sample) as described in the Metrics subsection.

	ft_recall	bt_recall	pt_recall	ft_precision	bt_precision	pt_precision
count	592.000000	592.000000	592.000000	592.000000	592.000000	592.000000
mean	0.548958	0.546847	0.463514	0.280405	0.276464	0.240428
std	0.428335	0.429380	0.428909	0.222395	0.219285	0.223212
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.500000	0.500000	0.500000	0.333333	0.333333	0.333333
75%	1.000000	1.000000	1.000000	0.333333	0.333333	0.333333
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

We observe that overall, the fastText model has the highest mean *recall@k'* and highest mean *precision@3*. The fastText model also has the lowest standard deviation of *recall@k'* while the BlazingText model has the lowest standard deviation of *precision@3*. These statistics suggest that the benchmark (fastText) model performs better than the other two proposed models, but only slightly better than the BlazingText model. Nevertheless, the quantitative results also suggest that more tuning can be needed for the two proposed models.

Furthermore, we observe that the quartiles are the same for all models and both *precision@3* and *recall@k'*, suggesting that the performance taking into account these metrics and these statistics are equal in quantity and we need to take into account other factors, such as more refined statistics or qualitative analysis.

b) Qualitative Results

In the following table, the prefixes *ft*, *bt* and *pt* stand for "fastText", "BlazingText" and "PyTorch" respectively. The *"true"* column contains the true tags for each sample.
(Note: the image can require zoom in the pdf file).

title	true	ft_predictions	bt_predictions	pt_predictions
Rendered pIxel width data for each character i...	[javascript, jquery, html, css]	[javascript, xml, python]	[xml, javascript, java]	[algorithm, c#, .net]
MVC architecture question for Mac application	[objective-c, cocoa, osx]	[flex, networking, actionscript-3]	[objective-c, flex, architecture]	[design-patterns, wpf, user-interface]
How could I get my SVN only host to pull from ...	[svn]	[svn, delphi, version-control]	[svn, delphi, ajax]	[.net, delphi, nhibernate]
C# vs Java generics	[c#, java, generics]	[java, networking, security]	[java, networking, exception]	[java, c#, c++]
NET Runtime 2 0 Error in a service	[.net, asp.net, .net-3.5, .net-2.0]	[.net, winforms, c#]	[.net, c#, .net-2.0]	[asp.net-mvc, c#, asp.net]
Implement validation as a method or as a property	[c#]	[c#, python, .net]	[.net, c#, python]	[.net, c#, xml]
Migrating Java UNO code from OpenOffice 2 4 to...	[java]	[java, eclipse, web-services]	[java, eclipse, javascript]	[unit-testing, java, c#]
How can I retrieve an assembly s qualified typ...	[c#, asp.net]	[.net, c#, exception]	[c#, .net, reflection]	[.net, c#, vb.net]
asp net sql timeout when not using sql	[c#, asp.net, sql-server]	[asp.net, c#, sql-server]	[asp.net, sql-server, sql]	[javascript, asp.net, jquery]
Dealing with C++ "initialized but not referenc...	[c++]	[c++, javascript, c#]	[c++, c#, javascript]	[c#, c++, .net]

The image shows only a pseudorandom sample of the test set with the true tags and predicted tags for each model, but after taking a look at several samples we note the following:

1) There are certain type of questions where each model gives right predictions. For example, those that contains the name of the programming language in the question title itself.

2) In most cases, the top-3 predictions made by the fastText and BlazingText models are very similar but not equal, which is expected since BlazingText is based on fastText but at the same time is interesting since BlazingText claims to have differences and enhancements over fastText.

3) Although the PyTorch's LSTM network has the lowest mean *precision@3* and *recall@k'*, it gives different but reasonable predictions in many cases. It is probable that this model performs better for certain kind of questions.

Overall, the points described above suggest that more qualitative analysis and comparison is needed.

V - Web Application & Conclusions

As stated in the Capstone Proposal for this Project, the final stage is **Hosting & Inference,** with the following steps:
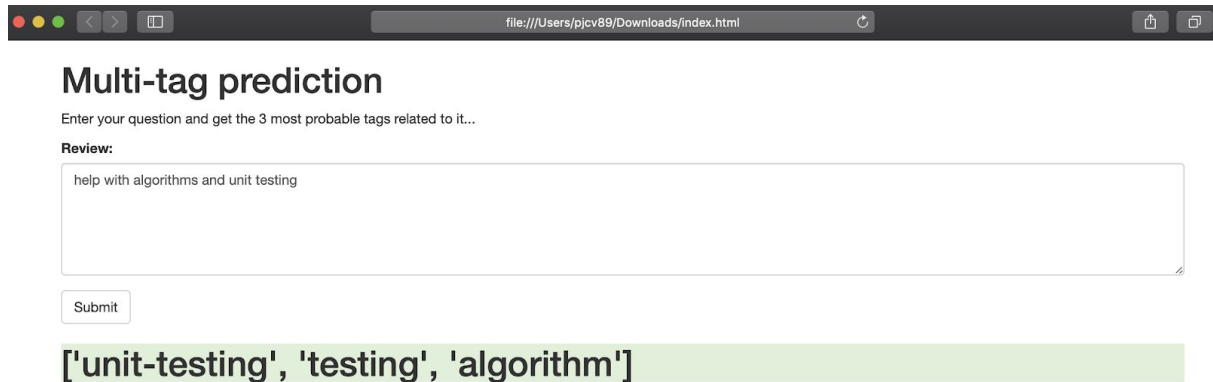
    i) Choose a model to deploy.

    ii) Setup a Lambda function via AWS Lambda and setup an API via Amazon API Gateway.

    iii) Build a web application and use it to perform real time inferences: given a question provided by the user, display its most probable tags according to the deployed model.

In this case, since the PyTorch's LSTM network was the most challenging model to develop since it implied to write custom training and prediction (serving) code, we wanted to finish this project by using it with the web app, so this was the chosen model to deploy. The web app is based on the html template provided for the Sentiment Analysis example (Part 3, Lesson 3 of this Nanodegree).

Now we show five examples by using the web app. In this toy app, we show the output as it is, as one string where we print the 3 most probable tags. In a real app, we would split each tag and show them properly to the user.

Examples of outputs of our web app.

Example 1: "help with algorithms and unit testing"

Multi-tag prediction

Enter your question and get the 3 most probable tags related to it...

Review:

help with algorithms and unit testing

Submit

['unit-testing', 'testing', 'algorithm']

Example 2: "I need help with ruby on rails"

Multi-tag prediction

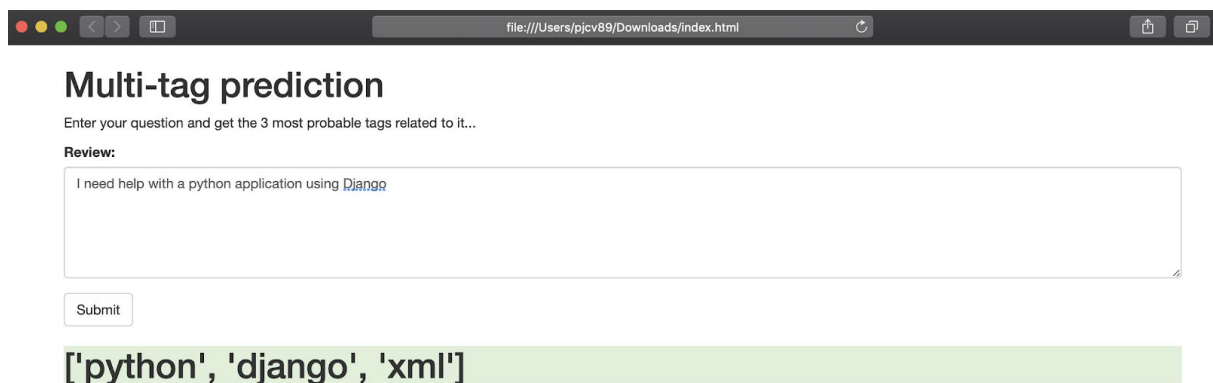Enter your question and get the 3 most probable tags related to it...

Review:

I need help with ruby on rails

Submit

['ruby-on-rails', 'ruby', 'java']

Example 3: "I need help with a python application using Django"

Multi-tag prediction

Enter your question and get the 3 most probable tags related to it...

Review:

I need help with a python application using Django

Submit

['python', 'django', 'xml']

Example 4: "help with sql 2005 queries"

Example 5: "how can I make Eclipse CDTR auto indent properly"

As **conclusion**, in this project we have:

- Investigated and applied several ways to deal with an **automatic tag generation** task, approaching the problem as a multi-label text classification from a ML perspective.

- Used a fastText model as baseline and compared quantitative and qualitative results with two proposed algorithms: BlazingText and a PyTorch's LSTM network.

- Wrote custom training and prediction (serving) code for the PyTorch and torchtext based model.

- Used the PyTorch and torchtext based model in a little web application using AWS tools.