Capstone Project Report
Machine Learning Engineer Nanodegree

Foxan Ng
5 January, 2019

Auto Tagging Stack Overflow Questions

Definition

Project Overview

Stack Overflow is the popular go-to resource from programming newbies to professionals. There is a joke that a programmer's job is to search for relevant code snippets on Stack Overflow to copy and paste. Over 18 million questions have been asked on this platform, and It is currently the largest question and answer site for various topics in computer programming.

For a platform that has such a high volume of data, it is essential for the questions to be tagged with relevant topics, such as python, apache-spark, c#, etc, so that it allows more effective search and shows the information to users of interest. Correctly tagging Stack Overflow questions can reduce both time to search for an answer and time a question got answered, and thus **increase overall user engagement**.

Natural language processing[1] is a domain of study to program computers to process and analyze large amounts of natural language data. Particularly, **document classification**[2] is a subdomain which deals with problems of assigning a document to one or more classes or categories, which is similar to the problem we are trying to solve. A number of relevant techniques include: **tf-idf** (term frequency-inverse document frequency), multiple-instance learning, latent semantic analysis, etc.

In this project, I have worked on a dataset made available by Stack Overflow, which it is released on the machine learning competition platform, Kaggle. It is named "StackSample: 10% of Stack Overflow Q&A"[3], which contains text from 10% of Stack Overflow questions and answers on programming topics. I have built a machine learning model to predict tags from question title and body text as input.

[1] Natural language processing
[2] Document classification
[3] StackSample: 10% of Stack Overflow Q&A

Problem Statement

The goal of this project is to create a **machine learning model that will predict tags for Stack Overflow questions**, i.e. given any question text as input, it will be tagged with **minimum of one and maximum of five most relevant topics**, such as javascript, sql, c#, (i.e. the same rule currently adopted by Stack Overflow).

For example:

Given the following input:

```
{
   "title": "ASP.NET Site Maps",
   "body": "Has anyone got experience creating SQL-based ASP.NET site-map providers?
I've got the default XML file web.sitemap working properly with my Menu and
SiteMapPath controls, but I'll need a way for the users of my site to create and
modify pages dynamically. I need to tie page viewing permissions into the standard
ASP.NET membership system as well."
}
```

The model will output:

```
{
   "tags": ["sql", "asp.net", "sitemap"]
}
```

Metrics

Hamming loss will be used as the evaluation metric, as it is a commonly used loss function in multilabel classification. It is basically the fraction of labels that are incorrectly predicted, so the smaller it is, the better the results are. Precision, recall and F_1 score will also be used to support as they are evaluation metrics used to understand model performance in general.

The definition of hamming loss[4] is the fraction of the wrong labels to the total number of labels,

$$\frac{1}{|N| \cdot |L|} \sum_{i=1}^{|N|} \sum_{j=1}^{|L|} \operatorname{xor}(y_{i,j}, z_{i,j})$$

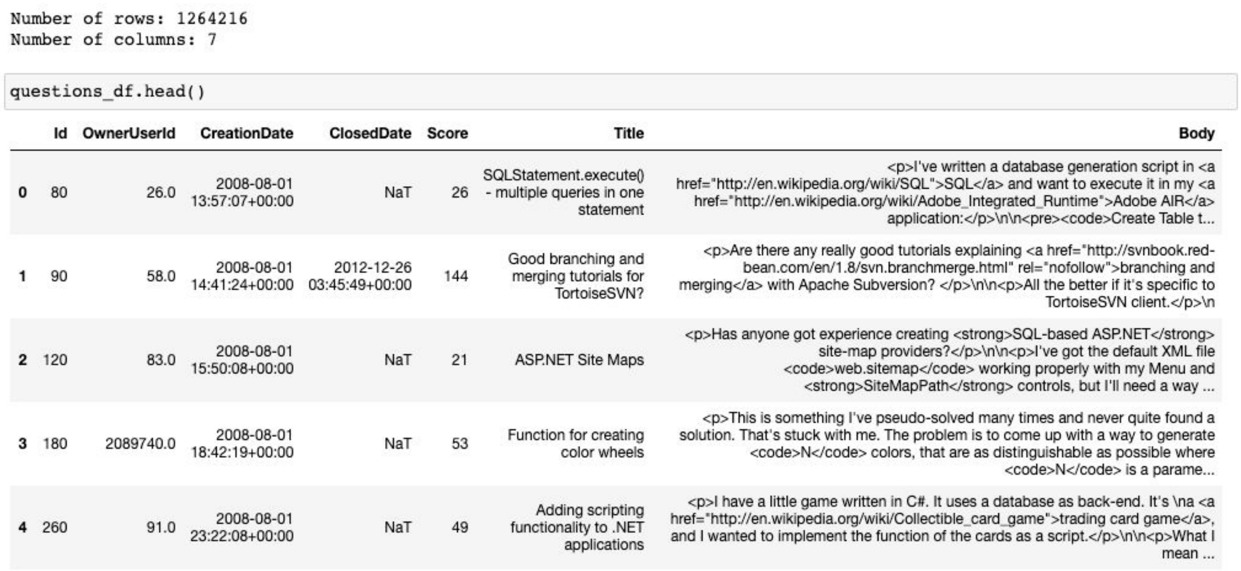
where $y_{i,j}$ is the target and $z_{i,j}$ is the prediction.
This is a loss function from 0 to 1, so the optimal value is 0.

[4] Multi-label classification

Analysis

Data Exploration and Visualization

Questions.csv



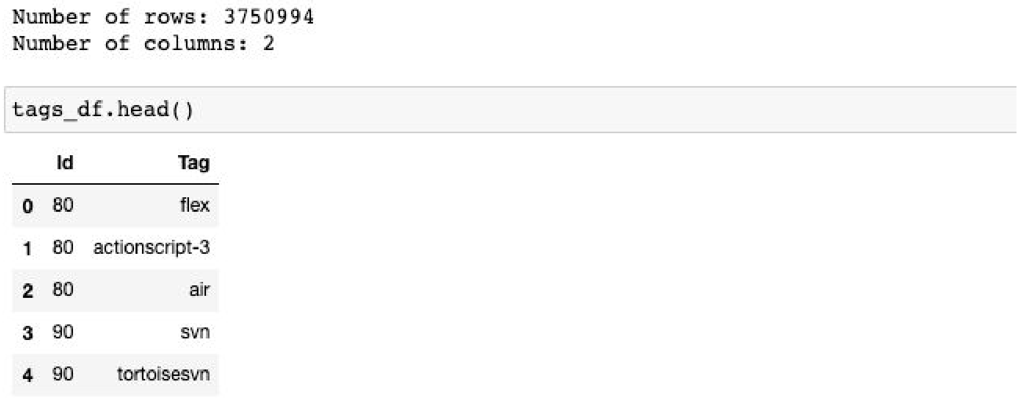
```
Number of rows: 1264216
Number of columns: 7
```

```
questions_df.head()
```

	Id	OwnerUserId	CreationDate	ClosedDate	Score	Title	Body
0	80	26.0	2008-08-01 13:57:07+00:00	NaT	26	SQLStatement.execute() - multiple queries in one statement	<p>I've written a database generation script in <a href="http://en.wikipedia.org/wiki/SQL">SQL</a> and want to execute it in my <a href="http://en.wikipedia.org/wiki/Adobe_Integrated_Runtime">Adobe AIR</a> application:</p>\n\n<pre><code>Create Table t...
1	90	58.0	2008-08-01 14:41:24+00:00	2012-12-26 03:45:49+00:00	144	Good branching and merging tutorials for TortoiseSVN?	<p>Are there any really good tutorials explaining <a href="http://svnbook.red-bean.com/en/1.8/svn.branchmerge.html" rel="nofollow">branching and merging</a> with Apache Subversion? </p>\n\n<p>All the better if it's specific to TortoiseSVN client.</p>\n
2	120	83.0	2008-08-01 15:50:08+00:00	NaT	21	ASP.NET Site Maps	<p>Has anyone got experience creating <strong>SQL-based ASP.NET</strong> site-map providers?</p>\n\n<p>I've got the default XML file <code>web.sitemap</code> working properly with my Menu and <strong>SiteMapPath</strong> controls, but I'll need a way ...
3	180	2089740.0	2008-08-01 18:42:19+00:00	NaT	53	Function for creating color wheels	<p>This is something I've pseudo-solved many times and never quite found a solution. That's stuck with me. The problem is to come up with a way to generate <code>N</code> colors, that are as distinguishable as possible where <code>N</code> is a parame...
4	260	91.0	2008-08-01 23:22:08+00:00	NaT	49	Adding scripting functionality to .NET applications	<p>I have a little game written in C#. It uses a database as back-end. It's \na <a href="http://en.wikipedia.org/wiki/Collectible_card_game">trading card game</a>, and I wanted to implement the function of the cards as a script.</p>\n\n<p>What I mean ...

Questions.csv contains **1.26 million questions**, created from 2008 August to 2016 October. We can see that Title is in plain text, while Body is in HTML format, which requires a lot of data cleansing before it is in a useful format. Also note that some symbols can be meaningful in this problem, e.g. ASP.NET, C#, etc, so we need to be careful not to remove them during data cleansing.

Tags.csv



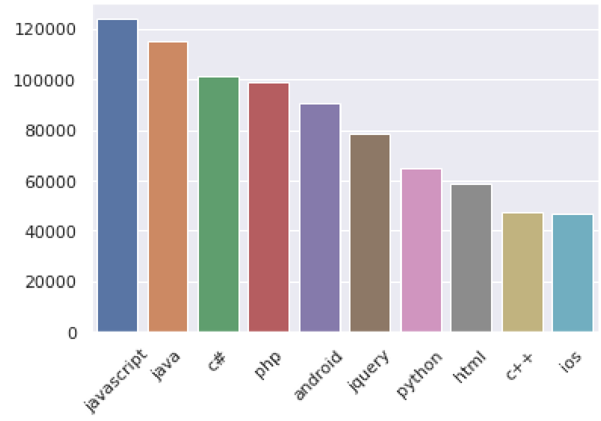
```
Number of rows: 3750994
Number of columns: 2
```

```
tags_df.head()
```

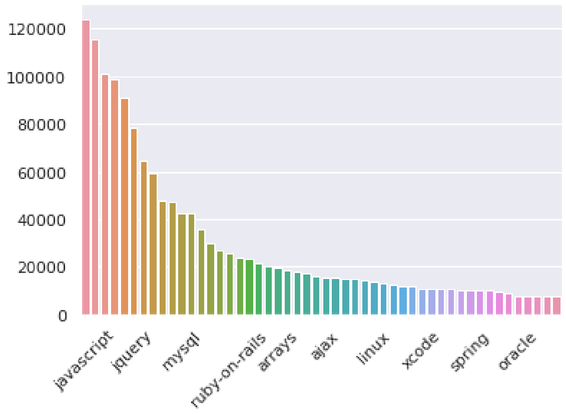
	Id	Tag
0	80	flex
1	80	actionscript-3
2	80	air
3	90	svn
4	90	tortoisesvn

Tags.csv contains over **3.75 million** Id (Question ID) and Tag pairs (one tag per row). It can be joined with the questions data using Id. There are more than **38,000 unique tags**.

Imbalanced classes



Top 10 tags



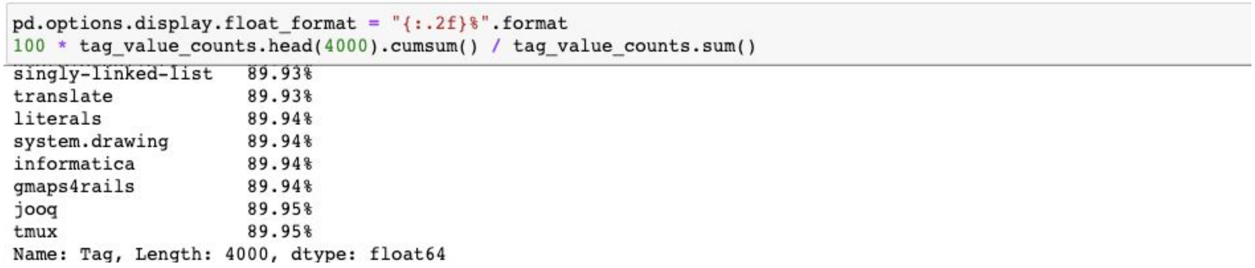
Top 50 tags

The top 10 tags contain a lot of popular programming language and framework, such as javascript, java and c#. We can already see that we have an imbalanced dataset here, for example, more than 10% of the questions are tagged with javascript.

Meanwhile, there are far more unpopular or new tags (such as pyqtgraph, slurm, geom-bar), which have very few questions (less than a hundred) asked.

As shown in the top 50 tags bar plot, the number of questions per tag clearly demonstrates a long tail distribution. Therefore, we can limit the number of tags to be included in the dataset, so that the model training can be more efficient, while still maintaining a high level of accuracy.

Number of classes to included



```
pd.options.display.float_format = "{:.2f}%".format
100 * tag_value_counts.head(4000).cumsum() / tag_value_counts.sum()
```

```
singly-linked-list    89.93%
translate             89.93%
literals              89.94%
system.drawing        89.94%
informatica           89.94%
gmaps4rails           89.94%
jooq                  89.95%
tmux                  89.95%
Name: Tag, Length: 4000, dtype: float64
```

The top 4000 tags cover almost 90% of the questions in the dataset. Therefore, I decided to limit the dataset to include only questions with the top 4000 tags to reduce the time for model training. We can always include more tags later in case we find the model is not as performant as expected. (4000 classes is still a very large number for a multilabel classification problem)

Algorithms and Techniques

For data cleansing, I have used **BeautifulSoup** to remove HTML tags from the body text. Natural Language Toolkit (**NLTK**) is then used for text processing on columns Title and Body, including removing stop words and word tokenization. Then, **scikit-learn** is used to extract useful features (**tf-idf**) from the cleansed dataframe.

As the problem is a multilabel classification problem, I have used **OneVsRestClassifier**[5] from scikit-learn to classify one tag versus all other tags at a time, and the model is trained by using **SGDClassifier**[6] as the estimator algorithm. More detailed implementation will be explained in the methodology section.

Benchmark

This problem has been tackled by others with a similar dataset[7], using OneVsRestClassifier and SGDClassifier with tf-idf as features. The author achieved a hamming loss of 0.002779, when they limited the number of topics to the 500 most popular ones in that dataset. I aim to achieve a similar or better performance for the final machine learning model built.

[5] sklearn.multiclass.OneVsRestClassifier — scikit-learn 0.22 documentation
[6] sklearn.linear_model.SGDClassifier — scikit-learn 0.22.1 documentation
[7] Predicting Tags for the Questions in Stack Overflow

Methodology

I have organized all crucial steps in a machine learning project into separate Jupyter notebooks, i.e. Exploratory Data Analysis, Text Processing, Feature Engineering and Model Training, for better understanding and easier to revisit a prior step in case we want to change some codes in the later stages. As we have already discussed data exploration in the last section, I will focus on the remaining three parts in this section.

Text Processing

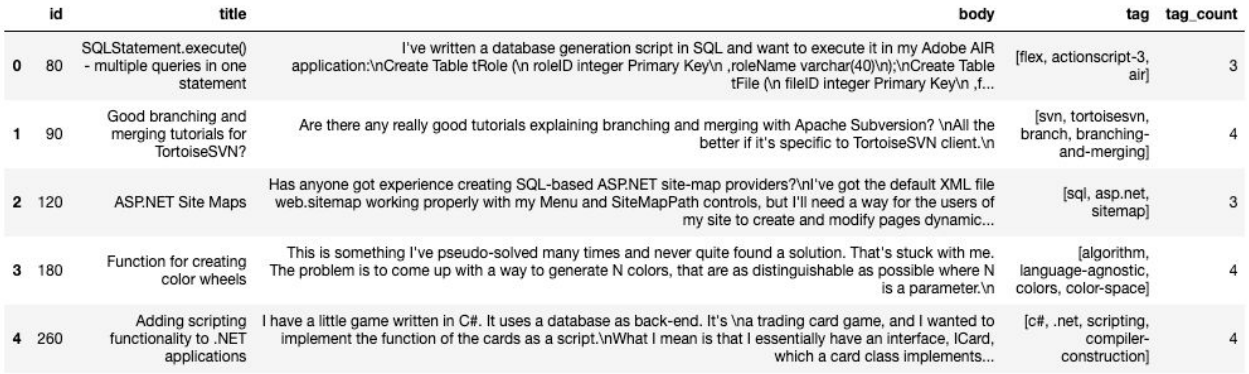
Data preprocessing and cleansing is the first and foremost step for every successful machine learning project, which is no exception to this project as well.

The dataset is first downloaded from Kaggle and unzipped. Pandas are then used to read the two csv file as dataframes:

```
questions_df = pd.read_csv(os.path.join(DATA_DIR, "Questions.csv"),
encoding="ISO-8859-1", parse_dates=["CreationDate", "ClosedDate"])
tags_df = pd.read_csv(os.path.join(DATA_DIR, "Tags.csv"), encoding="ISO-8859-1")
```

After some exploratory data analysis, the two dataframes are joined so that we have all tags in the same record. Then, BeautifulSoup with lxml parser is used to parse the body column to remove HTML tags.

```
df = questions_df[["id", "title", "body"]].merge(tags_per_question_df.to_frame(), on="id")
df["body"] = df["body"].progress_apply(lambda text: BeautifulSoup(text, "lxml").text)
```



	id	title	body	tag	tag_count
0	80	SQLStatement.execute() - multiple queries in one statement	I've written a database generation script in SQL and want to execute it in my Adobe AIR application:\nCreate Table tRole (\n roleID integer Primary Key\n ,roleName varchar(40)\n);\nCreate Table tFile (\n fileID integer Primary Key\n ,f...	[flex, actionscript-3, air]	3
1	90	Good branching and merging tutorials for TortoiseSVN?	Are there any really good tutorials explaining branching and merging with Apache Subversion? \nAll the better if it's specific to TortoiseSVN client.\n	[svn, tortoisesvn, branch, branching-and-merging]	4
2	120	ASP.NET Site Maps	Has anyone got experience creating SQL-based ASP.NET site-map providers?\nI've got the default XML file web.sitemap working properly with my Menu and SiteMapPath controls, but I'll need a way for the users of my site to create and modify pages dynamic...	[sql, asp.net, sitemap]	3
3	180	Function for creating color wheels	This is something I've pseudo-solved many times and never quite found a solution. That's stuck with me. The problem is to come up with a way to generate N colors, that are as distinguishable as possible where N is a parameter.\n	[algorithm, language-agnostic, colors, color-space]	4
4	260	Adding scripting functionality to .NET applications	I have a little game written in C#. It uses a database as back-end. It's \na trading card game, and I wanted to implement the function of the cards as a script.\nWhat I mean is that I essentially have an interface, ICard, which a card class implements...	[c#, .net, scripting, compiler-construction]	4

The dataset is then further standardized by converting text to lower cases, removing stop words and tokenizing the text using NLTK. I also have to keep a list of popular tags with symbols (e.g. node.js, objective-c, html5) to work around how NLTK tokenize words, as they are not considered legal words in NLTK:

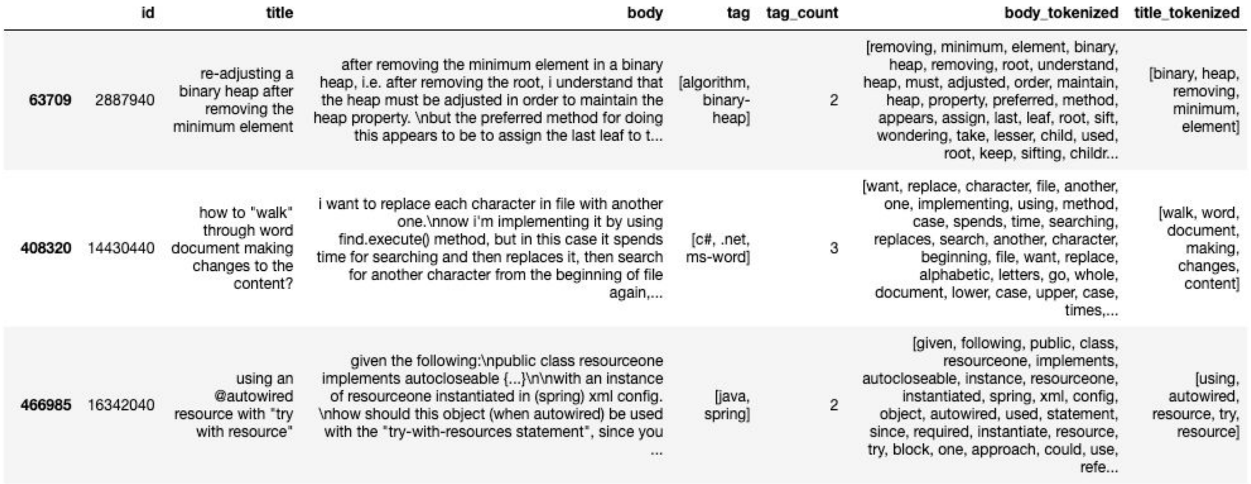
```
df["body"] = df["body"].str.lower()
topics_with_symbols = ["c#", "c++", ".net", "asp.net", "node.js", "objective-c", "unity3d", "html5", "css3",
                       "d3.js", "utf-8", "neo4j", "scikit-learn", "f#", "3d", "x86"]
df["body_tokenized"] = df["body"].progress_apply(lambda text: [word for word in nltk.word_tokenize(text) \
                                              if word.isalpha() or word in list("+#") + topics_with_symbols])
df["body_tokenized"] = df["body_tokenized"].progress_apply(filter_stop_words)
```

I then retokenized tokens such as ("c", "#"), ("c", "+", "+") back to "c#" and "c++". Although the machine learning model should still be able to pick up the original words as patterns, it is always better to handle them in advance:

```
mwe_tokenizer = nltk.MWETokenizer(separator="")
mwe_tokenizer.add_mwe(("c", "#"))
mwe_tokenizer.add_mwe(("c", "+", "+"))
df["body_tokenized"] = df["body_tokenized"].progress_apply(lambda tokens: [token for
token in mwe_tokenizer.tokenize(tokens)])
```

The dataset is then filtered to only include top 4000 tags.

The interim result after text processing:

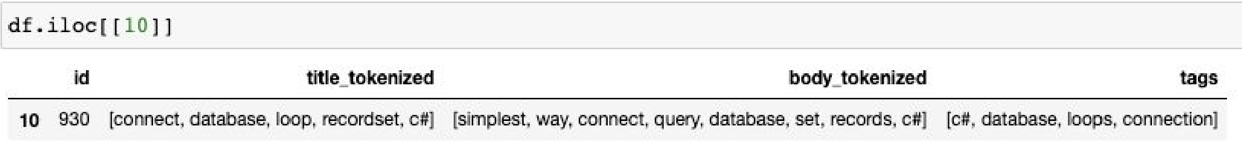


	id	title	body	tag	tag_count	body_tokenized	title_tokenized
63709	2887940	re-adjusting a binary heap after removing the minimum element	after removing the minimum element in a binary heap, i.e. after removing the root, i understand that the heap must be adjusted in order to maintain the heap property. \nbut the preferred method for doing this appears to be to assign the last leaf to t...	[algorithm, binary-heap]	2	[removing, minimum, element, binary, heap, removing, root, understand, heap, must, adjusted, order, maintain, heap, property, preferred, method, appears, assign, last, leaf, root, sift, wondering, take, lesser, child, used, root, keep, sifting, childr...	[binary, heap, removing, minimum, element]
408320	14430440	how to "walk" through word document making changes to the content?	i want to replace each character in file with another one.\nnow i'm implementing it by using find.execute() method, but in this case it spends time for searching and then replaces it, then search for another character from the beginning of file again,...	[c#, .net, ms-word]	3	[want, replace, character, file, another, one, implementing, using, method, case, spends, time, searching, replaces, search, another, character, beginning, file, want, replace, alphabetic, letters, go, whole, document, lower, case, upper, case, times,...	[walk, word, document, making, changes, content]
466985	16342040	using an @autowired resource with "try with resource"	given the following:\npublic class resourceone implements autocloseable {...}\n\nwith an instance of resourceone instantiated in (spring) xml config. \nhow should this object (when autowired) be used with the "try-with-resources statement", since you ...	[java, spring]	2	[given, following, public, class, resourceone, implements, autocloseable, instance, resourceone, instantiated, spring, xml, config, object, autowired, used, statement, since, required, instantiate, resource, try, block, one, approach, could, use, refe...	[using, autowired, resource, try, resource]

Feature Engineering

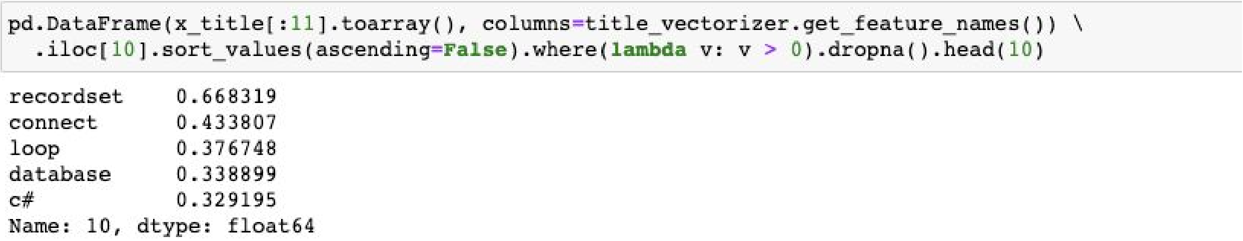
TfidfVectorizer[8] from scikit-learn is used to extract term frequency-inverse document frequency (tf-idf) as features from title and body. Tf-idf[9] is a numeric statistic to reflect how important a word is to a document in a collection, which is very useful in document classification:

```
from sklearn.feature_extraction.text import TfidfVectorizer
# we have already tokenize the text so we need a dummy one to bypass tokenization
def dummy_tokenizer(string): return string
# we will only get the 10,000 most common words for title to limit size of dataset
title_vectorizer = TfidfVectorizer(tokenizer=dummy_tokenizer, lowercase=False,
max_features=10000)
x_title = title_vectorizer.fit_transform(df["title_tokenized"])
# we will get the 100,000 most common words for body
body_vectorizer = TfidfVectorizer(tokenizer=dummy_tokenizer, lowercase=False,
max_features=100000)
x_body = body_vectorizer.fit_transform(df["body_tokenized"])
```



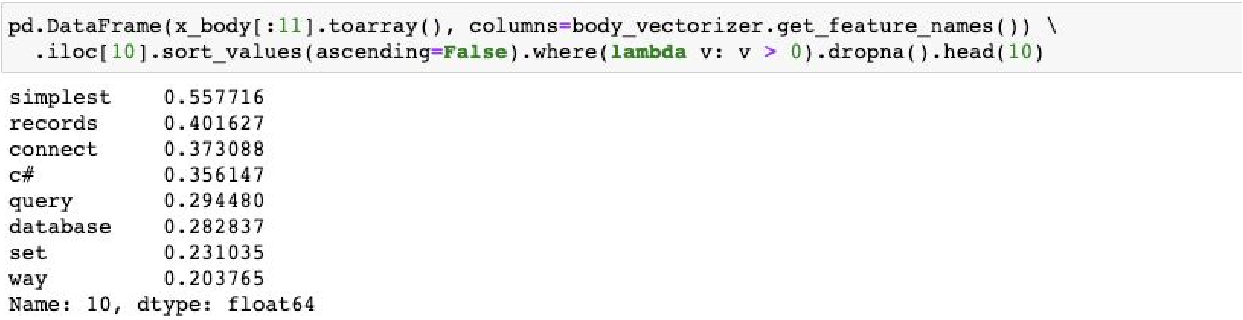
```
df.iloc[[10]]
```

	id	title_tokenized	body_tokenized	tags
10	930	[connect, database, loop, recordset, c#]	[simplest, way, connect, query, database, set, records, c#]	[c#, database, loops, connection]



```
pd.DataFrame(x_title[:11].toarray(), columns=title_vectorizer.get_feature_names()) \
  .iloc[10].sort_values(ascending=False).where(lambda v: v > 0).dropna().head(10)
```

```
recordset    0.668319
connect      0.433807
loop         0.376748
database     0.338899
c#           0.329195
Name: 10, dtype: float64
```



```
pd.DataFrame(x_body[:11].toarray(), columns=body_vectorizer.get_feature_names()) \
  .iloc[10].sort_values(ascending=False).where(lambda v: v > 0).dropna().head(10)
```

```
simplest    0.557716
records     0.401627
connect     0.373088
c#          0.356147
query       0.294480
database    0.282837
set         0.231035
way         0.203765
Name: 10, dtype: float64
```

[8] sklearn.feature_extraction.text.TfidfVectorizer — scikit-learn 0.22.1 documentation
[9] tf–idf

I have included 10,000 most common words from title and 100,000 most common words form body as features, i.e. there are 110,000 features in total.

From the example above, we can see keywords from the features, like connect, loop, c# and database, which are similar to the actual tags.

As title is much more representative in terms of understanding the question, I have given a weight of 2 to features derived from it during feature engineering. The labels (i.e. tags) are then transformed from the intuitive format (e.g. [“c#”, “.net”, “ms-word”]) into an array of 4000 binary classes so that it can be fit for model training:

```
from scipy.sparse import hstack
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MultiLabelBinarizer

x_title = x_title * 2
X = hstack([x_title, x_body])
y = df[["tags"]]

multi_label_binarizer = MultiLabelBinarizer()
y = multi_label_binarizer.fit_transform(y["tags"])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 0)
```

Model Training

Initially, I have decided to use SageMaker to train models, as it offers some useful features, such as autoscaling, hyperparameter tuning, etc. Unfortunately, SageMaker does not have built-in algorithm for multilabel classification. Although it offers support for scikit-learn models, as scikit-learn only runs on a single CPU-only machine, we cannot make use of autoscaling from SageMaker, so I decided to abandon using SageMaker for model training.

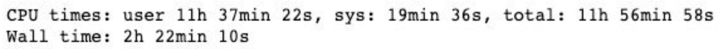
As this is a multilabel classification, I used **OneVsRestClassifier** from scikit-learn to fit one classifier per class (i.e. 4,000 classifiers will be trained in total, where each class is fitted against all other classes). I first decided to use XGBClassifier as the underlying algorithm for classification:

```
from sklearn.multiclass import OneVsRestClassifier
from xgboost import XGBClassifier
```

```
xgb_classifier = XGBClassifier(max_depth=5, eta=0.2, gamma=4, min_child_weight=6,
                  subsample=0.8, early_stopping_rounds=10, num_round=200, n_jobs=-1)

clf = OneVsRestClassifier(xgb_classifier)
clf.fit(X_train, y_train)
```

However, it takes way too long (more than an hour) to train even on a much smaller dataset (10,000 rows, ~1% of total), so I decided to switch to linear model using **SGDClassifier**:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.linear_model import SGDClassifier

sgd_classifier = SGDClassifier(n_jobs=-1)

clf = OneVsRestClassifier(sgd_classifier)
clf.fit(X_train, y_train)
```



```
CPU times: user 11h 37min 22s, sys: 19min 36s, total: 11h 56min 58s
Wall time: 2h 22min 10s
```

Nevertheless, it still took around 5 hours to train the model (which actually contains 4,000 models, one model for each class) on a whopping ml.m5.12xlarge instance on AWS. At this point, it is too costly to perform hyperparameter tuning to reach a better model.

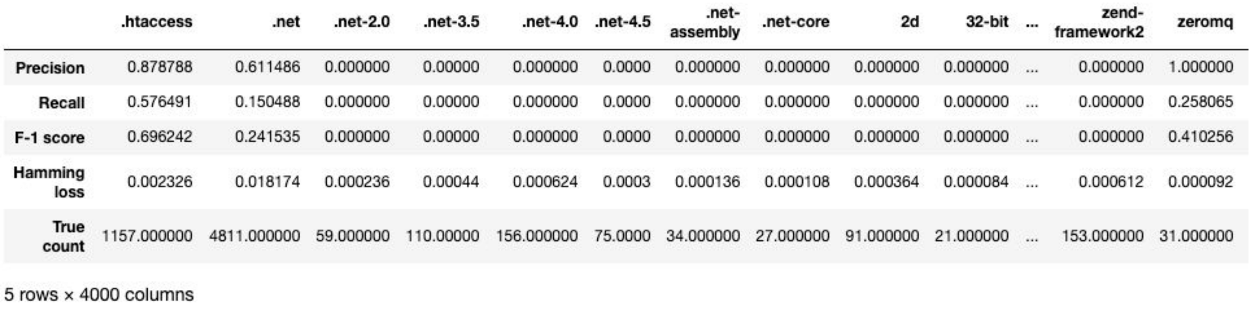
Results

Model Evaluation and Validation

The final model is evaluated using hamming loss, which is defined as the fraction of labels that are incorrectly predicted. I have also included other common metrics, such as precision, recall and F_1 score so that we can have a better understanding on the model performance:

```
from sklearn.metrics import precision_recall_fscore_support as score
from sklearn.metrics import hamming_loss

precision, recall, fscore, support = score(y_test, y_pred)
hamming = []
for i, (test, pred) in enumerate(zip(y_test.T, y_pred.T)):
   hamming.append(hamming_loss(test, pred))

metric_df = pd.DataFrame(data=[precision, recall, fscore, hamming, support],
              index=["Precision", "Recall", "F-1 score", "Hamming loss", "True count"], columns=y_classes)
```

All 4000 tags

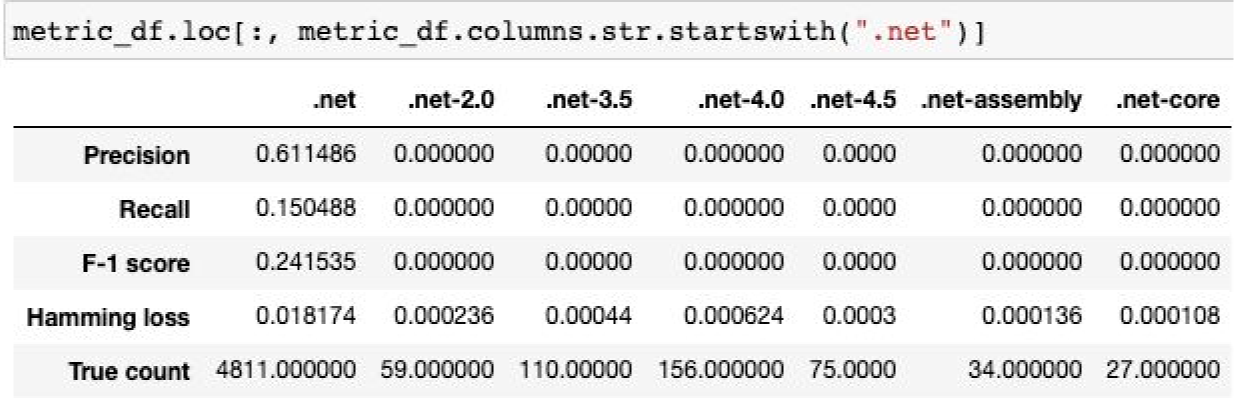


	.htaccess	.net	.net-2.0	.net-3.5	.net-4.0	.net-4.5	.net-assembly	.net-core	2d	32-bit	...	zend-framework2	zeromq
Precision	0.878788	0.611486	0.000000	0.00000	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	...	0.000000	1.000000
Recall	0.576491	0.150488	0.000000	0.00000	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.258065
F-1 score	0.696242	0.241535	0.000000	0.00000	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.410256
Hamming loss	0.002326	0.018174	0.000236	0.00044	0.000624	0.0003	0.000136	0.000108	0.000364	0.000084	...	0.000612	0.000092
True count	1157.000000	4811.000000	59.000000	110.00000	156.000000	75.0000	34.000000	27.000000	91.000000	21.000000	...	153.000000	31.000000

5 rows × 4000 columns

Just by a quick glance, we can see that the results for some tags are quite good, e.g. .htaccess, zookeeper, zsh. The overall results are not as good as expected, which is because there are a lot of tags that results in no prediction at all.

Tags contained .net (as an example)



```
metric_df.loc[:, metric_df.columns.str.startswith(".net")]
```

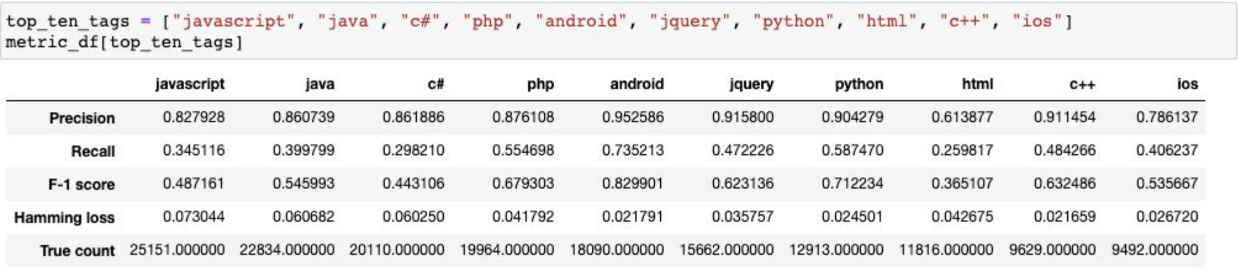
	.net	.net-2.0	.net-3.5	.net-4.0	.net-4.5	.net-assembly	.net-core
Precision	0.611486	0.000000	0.00000	0.000000	0.0000	0.000000	0.000000
Recall	0.150488	0.000000	0.00000	0.000000	0.0000	0.000000	0.000000
F-1 score	0.241535	0.000000	0.00000	0.000000	0.0000	0.000000	0.000000
Hamming loss	0.018174	0.000236	0.00044	0.000624	0.0003	0.000136	0.000108
True count	4811.000000	59.000000	110.00000	156.000000	75.0000	34.000000	27.000000

For example, consider the list of tags with .net in the name. We can see that the reason that a lot of tags has no predictions, is because:

- they have too few examples (True count) in the dataset.
- they are being tagged for the main topic instead of the sub topics.

e.g. specific version of .net (e.g. .net-2.0, .net-3.5), or sub topics under .net (e.g. .net-assembly, .net-core). We can filter them out to see how the model actually performs for major tags.

Top 10 tags



```
top_ten_tags = ["javascript", "java", "c#", "php", "android", "jquery", "python", "html", "c++", "ios"]
metric_df[top_ten_tags]
```

	javascript	java	c#	php	android	jquery	python	html	c++	ios
Precision	0.827928	0.860739	0.861886	0.876108	0.952586	0.915800	0.904279	0.613877	0.911454	0.786137
Recall	0.345116	0.399799	0.298210	0.554698	0.735213	0.472226	0.587470	0.259817	0.484266	0.406237
F-1 score	0.487161	0.545993	0.443106	0.679303	0.829901	0.623136	0.712234	0.365107	0.632486	0.535667
Hamming loss	0.073044	0.060682	0.060250	0.041792	0.021791	0.035757	0.024501	0.042675	0.021659	0.026720
True count	25151.000000	22834.000000	20110.000000	19964.000000	18090.000000	15662.000000	12913.000000	11816.000000	9629.000000	9492.000000

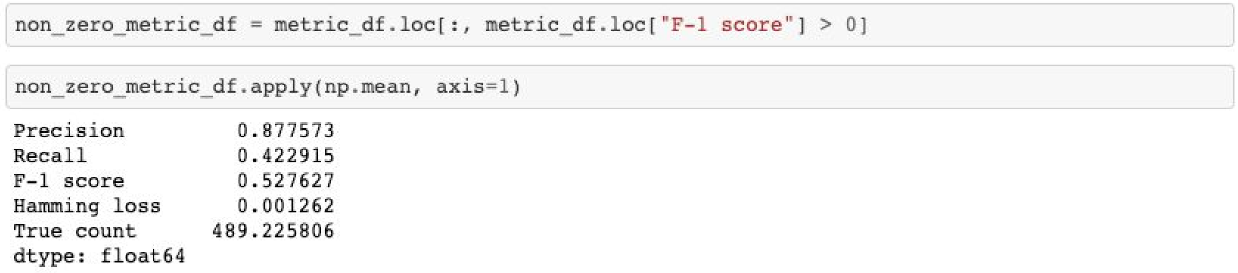
The metrics for the top 10 tags are actually pretty good. It is partly because they have a lot of samples (true count) in the dataset. The best one is android while the worst one is html. This makes sense because a lot of questions may contain the word “html” (which is common in code snippet), but the questions may not be related to the topic html at all.

Justification

We cannot get a good picture on how the models perform overall, as a lot of classes have too few examples in the dataset, resulting in no predictions, even though we have limited it to the top 4000 tags.

In retrospect, we can further limit the dataset to top 1000 tags before training the model, or perform some graph analysis to have a better understanding among topics (e.g. .net) and subtopics (.net-assembly) within the tags, and apply filters accordingly. The problem may then be solved by other approaches, such as Hierarchical Classification.

To work around for now, we can filter out the classes with no prediction at all and see how the models perform:



```
non_zero_metric_df = metric_df.loc[:, metric_df.loc["F-1 score"] > 0]
```

```
non_zero_metric_df.apply(np.mean, axis=1)
```

```
Precision         0.877573
Recall            0.422915
F-1 score         0.527627
Hamming loss      0.001262
True count      489.225806
dtype: float64
```

There are 930 tags with predictions from this model, and the average hamming loss is 0.001262, which beats the benchmark (0.002779), which contains only the top 500 popular tags. Moreover, it has a precision of 0.8775 and a recall of 0.4229. We can conclude that this model is capable of predicting more tags with a lower loss.