**Predict tags on StackOverflow with linear models**

In this assignment you will learn how to predict tags for posts from [StackOverflow](https://stackoverflow.com/). To solve this task you will use multilabel classification approach.

**Libraries**

In this task you will need the following libraries:

* [Numpy](http://www.numpy.org/) — a package for scientific computing.
* [Pandas](https://pandas.pydata.org/) — a library providing high-performance, easy-to-use data structures and data analysis tools for the Python
* [scikit-learn](http://scikit-learn.org/stable/index.html) — a tool for data mining and data analysis.
* [NLTK](http://www.nltk.org/) — a platform to work with natural language.

**Data**

The following cell will download all data required for this assignment into the folder week1/data.

In [ ]:

**import** **sys**

sys.path.append("..")

**from** **common.download\_utils** **import** download\_week1\_resources

download\_week1\_resources()

**Text preprocessing**

For this and most of the following assignments you will need to use a list of stop words. It can be downloaded from *nltk*:

In [ ]:

**import** **nltk**

nltk.download('stopwords')

**from** **nltk.corpus** **import** stopwords

In this task you will deal with a dataset of post titles from StackOverflow. You are provided a split to 3 sets: *train*, *validation* and *test*. All corpora (except for *test*) contain titles of the posts and corresponding tags (100 tags are available). The *test* set is provided for Coursera's grading and doesn't contain answers. Upload the corpora using *pandas* and look at the data:

In [ ]:

**from** **ast** **import** literal\_eval

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

In [ ]:

**def** read\_data(filename):

data = pd.read\_csv(filename, sep='**\t**')

data['tags'] = data['tags'].apply(literal\_eval)

**return** data

In [ ]:

train = read\_data('data/train.tsv')

validation = read\_data('data/validation.tsv')

test = pd.read\_csv('data/test.tsv', sep='**\t**')

In [ ]:

train.head()

As you can see, *title* column contains titles of the posts and *tags* column contains the tags. It could be noticed that a number of tags for a post is not fixed and could be as many as necessary.

For a more comfortable usage, initialize *X\_train*, *X\_val*, *X\_test*, *y\_train*, *y\_val*.

In [ ]:

X\_train, y\_train = train['title'].values, train['tags'].values

X\_val, y\_val = validation['title'].values, validation['tags'].values

X\_test = test['title'].values

One of the most known difficulties when working with natural data is that it's unstructured. For example, if you use it "as is" and extract tokens just by splitting the titles by whitespaces, you will see that there are many "weird" tokens like *3.5?*, *"Flip*, etc. To prevent the problems, it's usually useful to prepare the data somehow. In this task you'll write a function, which will be also used in the other assignments.

**Task 1 (TextPrepare).** Implement the function *text\_prepare* following the instructions. After that, run the function *test\_test\_prepare* to test it on tiny cases and submit it to Coursera.

In [ ]:

**import** **re**

In [ ]:

REPLACE\_BY\_SPACE\_RE = re.compile('[/()**{}**\[\]\|@,;]')

BAD\_SYMBOLS\_RE = re.compile('[^0-9a-z #+\_]')

STOPWORDS = set(stopwords.words('english'))

**def** text\_prepare(text):

*"""*

*text: a string*

*return: modified initial string*

*"""*

text = *# lowercase text*

text = *# replace REPLACE\_BY\_SPACE\_RE symbols by space in text*

text = *# delete symbols which are in BAD\_SYMBOLS\_RE from text*

text = *# delete stopwords from text*

**return** text

In [ ]:

**def** test\_text\_prepare():

examples = ["SQL Server - any equivalent of Excel's CHOOSE function?",

"How to free c++ memory vector<int> \* arr?"]

answers = ["sql server equivalent excels choose function",

"free c++ memory vectorint arr"]

**for** ex, ans **in** zip(examples, answers):

**if** text\_prepare(ex) != ans:

**return** "Wrong answer for the case: '**%s**'" % ex

**return** 'Basic tests are passed.'

In [ ]:

print(test\_text\_prepare())

Run your implementation for questions from file *text\_prepare\_tests.tsv* to earn the points.

In [ ]:

prepared\_questions = []

**for** line **in** open('data/text\_prepare\_tests.tsv', encoding='utf-8'):

line = text\_prepare(line.strip())

prepared\_questions.append(line)

text\_prepare\_results = '**\n**'.join(prepared\_questions)

grader.submit\_tag('TextPrepare', text\_prepare\_results)

Now we can preprocess the titles using function *text\_prepare* and making sure that the headers don't have bad symbols:

In [ ]:

X\_train = [text\_prepare(x) **for** x **in** X\_train]

X\_val = [text\_prepare(x) **for** x **in** X\_val]

X\_test = [text\_prepare(x) **for** x **in** X\_test]

In [ ]:

X\_train[:3]

For each tag and for each word calculate how many times they occur in the train corpus.

**Task 2 (WordsTagsCount).** Find 3 most popular tags and 3 most popular words in the train data and submit the results to earn the points.

In [ ]:

*# Dictionary of all tags from train corpus with their counts.*

tags\_counts = {}

*# Dictionary of all words from train corpus with their counts.*

words\_counts = {}

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

We are assuming that *tags\_counts* and *words\_counts* are dictionaries like {'some\_word\_or\_tag': frequency}. After applying the sorting procedure, results will be look like this: [('most\_popular\_word\_or\_tag', frequency), ('less\_popular\_word\_or\_tag', frequency), ...]. The grader gets the results in the following format (two comma-separated strings with line break):

tag1,tag2,tag3

word1,word2,word3

Pay attention that in this assignment you should not submit frequencies or some additional information.

In [ ]:

most\_common\_tags = sorted(tags\_counts.items(), key=**lambda** x: x[1], reverse=**True**)[:3]

most\_common\_words = sorted(words\_counts.items(), key=**lambda** x: x[1], reverse=**True**)[:3]

grader.submit\_tag('WordsTagsCount', '**%s\n%s**' % (','.join(tag **for** tag, \_ **in** most\_common\_tags),

','.join(word **for** word, \_ **in** most\_common\_words)))

**Transforming text to a vector**

Machine Learning algorithms work with numeric data and we cannot use the provided text data "as is". There are many ways to transform text data to numeric vectors. In this task you will try to use two of them.

**Bag of words**

One of the well-known approaches is a *bag-of-words* representation. To create this transformation, follow the steps:

1. Find *N* most popular words in train corpus and numerate them. Now we have a dictionary of the most popular words.
2. For each title in the corpora create a zero vector with the dimension equals to *N*.
3. For each text in the corpora iterate over words which are in the dictionary and increase by 1 the corresponding coordinate.

Let's try to do it for a toy example. Imagine that we have *N* = 4 and the list of the most popular words is

['hi', 'you', 'me', 'are']

Then we need to numerate them, for example, like this:

{'hi': 0, 'you': 1, 'me': 2, 'are': 3}

And we have the text, which we want to transform to the vector:

'hi how are you'

For this text we create a corresponding zero vector

[0, 0, 0, 0]

And iterate over all words, and if the word is in the dictionary, we increase the value of the corresponding position in the vector:

'hi': [1, 0, 0, 0]

'how': [1, 0, 0, 0] # word 'how' is not in our dictionary

'are': [1, 0, 0, 1]

'you': [1, 1, 0, 1]

The resulting vector will be

[1, 1, 0, 1]

Implement the described encoding in the function *my\_bag\_of\_words* with the size of the dictionary equals to 5000. To find the most common words use train data. You can test your code using the function *test\_my\_bag\_of\_words*.

In [ ]:

DICT\_SIZE = 5000

WORDS\_TO\_INDEX = *####### YOUR CODE HERE #######*

INDEX\_TO\_WORDS = *####### YOUR CODE HERE #######*

ALL\_WORDS = WORDS\_TO\_INDEX.keys()

**def** my\_bag\_of\_words(text, words\_to\_index, dict\_size):

*"""*

*text: a string*

*dict\_size: size of the dictionary*

*return a vector which is a bag-of-words representation of 'text'*

*"""*

result\_vector = np.zeros(dict\_size)

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

**return** result\_vector

In [ ]:

**def** test\_my\_bag\_of\_words():

words\_to\_index = {'hi': 0, 'you': 1, 'me': 2, 'are': 3}

examples = ['hi how are you']

answers = [[1, 1, 0, 1]]

**for** ex, ans **in** zip(examples, answers):

**if** (my\_bag\_of\_words(ex, words\_to\_index, 4) != ans).any():

**return** "Wrong answer for the case: '**%s**'" % ex

**return** 'Basic tests are passed.'

In [ ]:

print(test\_my\_bag\_of\_words())

Now apply the implemented function to all samples (this might take up to a minute):

In [ ]:

**from** **scipy** **import** sparse **as** sp\_sparse

In [ ]:

X\_train\_mybag = sp\_sparse.vstack([sp\_sparse.csr\_matrix(my\_bag\_of\_words(text, WORDS\_TO\_INDEX, DICT\_SIZE)) **for** text **in** X\_train])

X\_val\_mybag = sp\_sparse.vstack([sp\_sparse.csr\_matrix(my\_bag\_of\_words(text, WORDS\_TO\_INDEX, DICT\_SIZE)) **for** text **in** X\_val])

X\_test\_mybag = sp\_sparse.vstack([sp\_sparse.csr\_matrix(my\_bag\_of\_words(text, WORDS\_TO\_INDEX, DICT\_SIZE)) **for** text **in** X\_test])

print('X\_train shape ', X\_train\_mybag.shape)

print('X\_val shape ', X\_val\_mybag.shape)

print('X\_test shape ', X\_test\_mybag.shape)

As you might notice, we transform the data to sparse representation, to store the useful information efficiently. There are many [types](https://docs.scipy.org/doc/scipy/reference/sparse.html) of such representations, however sklearn algorithms can work only with [csr](https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html" \l "scipy.sparse.csr_matrix) matrix, so we will use this one.

**Task 3 (BagOfWords).** For the 11th row in *X\_train\_mybag* find how many non-zero elements it has. In this task the answer (variable *non\_zero\_elements\_count*) should be a number, e.g. 20.

In [ ]:

row = X\_train\_mybag[10].toarray()[0]

non\_zero\_elements\_count = *####### YOUR CODE HERE #######*

grader.submit\_tag('BagOfWords', str(non\_zero\_elements\_count))

**TF-IDF**

The second approach extends the bag-of-words framework by taking into account total frequencies of words in the corpora. It helps to penalize too frequent words and provide better features space.

Implement function *tfidf\_features* using class [TfidfVectorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) from *scikit-learn*. Use *train* corpus to train a vectorizer. Don't forget to take a look into the arguments that you can pass to it. We suggest that you filter out too rare words (occur less than in 5 titles) and too frequent words (occur more than in 90% of the titles). Also, use bigrams along with unigrams in your vocabulary.

In [ ]:

**from** **sklearn.feature\_extraction.text** **import** TfidfVectorizer

In [ ]:

**def** tfidf\_features(X\_train, X\_val, X\_test):

*"""*

*X\_train, X\_val, X\_test — samples*

*return TF-IDF vectorized representation of each sample and vocabulary*

*"""*

*# Create TF-IDF vectorizer with a proper parameters choice*

*# Fit the vectorizer on the train set*

*# Transform the train, test, and val sets and return the result*

tfidf\_vectorizer = *####### YOUR CODE HERE #######*

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

**return** X\_train, X\_val, X\_test, tfidf\_vectorizer.vocabulary\_

Once you have done text preprocessing, always have a look at the results. Be very careful at this step, because the performance of future models will drastically depend on it.

In this case, check whether you have c++ or c# in your vocabulary, as they are obviously important tokens in our tags prediction task:

In [ ]:

X\_train\_tfidf, X\_val\_tfidf, X\_test\_tfidf, tfidf\_vocab = tfidf\_features(X\_train, X\_val, X\_test)

tfidf\_reversed\_vocab = {i:word **for** word,i **in** tfidf\_vocab.items()}

In [ ]:

*######### YOUR CODE HERE #############*

If you can't find it, we need to understand how did it happen that we lost them? It happened during the built-in tokenization of TfidfVectorizer. Luckily, we can influence on this process. Get back to the function above and use '(\S+)' regexp as a *token\_pattern* in the constructor of the vectorizer.

Now, use this transormation for the data and check again.

In [ ]:

*######### YOUR CODE HERE #############*

**MultiLabel classifier**

As we have noticed before, in this task each example can have multiple tags. To deal with such kind of prediction, we need to transform labels in a binary form and the prediction will be a mask of 0s and 1s. For this purpose it is convenient to use [MultiLabelBinarizer](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MultiLabelBinarizer.html) from *sklearn*.

In [ ]:

**from** **sklearn.preprocessing** **import** MultiLabelBinarizer

In [ ]:

mlb = MultiLabelBinarizer(classes=sorted(tags\_counts.keys()))

y\_train = mlb.fit\_transform(y\_train)

y\_val = mlb.fit\_transform(y\_val)

Implement the function *train\_classifier* for training a classifier. In this task we suggest to use One-vs-Rest approach, which is implemented in [OneVsRestClassifier](http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html) class. In this approach *k* classifiers (= number of tags) are trained. As a basic classifier, use [LogisticRegression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). It is one of the simplest methods, but often it performs good enough in text classification tasks. It might take some time, because a number of classifiers to train is large.

In [ ]:

**from** **sklearn.multiclass** **import** OneVsRestClassifier

**from** **sklearn.linear\_model** **import** LogisticRegression, RidgeClassifier

In [ ]:

**def** train\_classifier(X\_train, y\_train):

*"""*

*X\_train, y\_train — training data*

*return: trained classifier*

*"""*

*# Create and fit LogisticRegression wraped into OneVsRestClassifier.*

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

Train the classifiers for different data transformations: *bag-of-words* and *tf-idf*.

In [ ]:

classifier\_mybag = train\_classifier(X\_train\_mybag, y\_train)

classifier\_tfidf = train\_classifier(X\_train\_tfidf, y\_train)

Now you can create predictions for the data. You will need two types of predictions: labels and scores.

In [ ]:

y\_val\_predicted\_labels\_mybag = classifier\_mybag.predict(X\_val\_mybag)

y\_val\_predicted\_scores\_mybag = classifier\_mybag.decision\_function(X\_val\_mybag)

y\_val\_predicted\_labels\_tfidf = classifier\_tfidf.predict(X\_val\_tfidf)

y\_val\_predicted\_scores\_tfidf = classifier\_tfidf.decision\_function(X\_val\_tfidf)

Now take a look at how classifier, which uses TF-IDF, works for a few examples:

In [ ]:

y\_val\_pred\_inversed = mlb.inverse\_transform(y\_val\_predicted\_labels\_tfidf)

y\_val\_inversed = mlb.inverse\_transform(y\_val)

**for** i **in** range(3):

print('Title:**\t{}\n**True labels:**\t{}\n**Predicted labels:**\t{}\n\n**'.format(

X\_val[i],

','.join(y\_val\_inversed[i]),

','.join(y\_val\_pred\_inversed[i])

))

Now, we would need to compare the results of different predictions, e.g. to see whether TF-IDF transformation helps or to try different regularization techniques in logistic regression. For all these experiments, we need to setup evaluation procedure.

**Evaluation**

To evaluate the results we will use several classification metrics:

* [Accuracy](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html)
* [F1-score](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html)
* [Area under ROC-curve](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html)
* [Area under precision-recall curve](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html#sklearn.metrics.average_precision_score)

Make sure you are familiar with all of them. How would you expect the things work for the multi-label scenario? Read about micro/macro/weighted averaging following the sklearn links provided above.

In [ ]:

**from** **sklearn.metrics** **import** accuracy\_score

**from** **sklearn.metrics** **import** f1\_score

**from** **sklearn.metrics** **import** roc\_auc\_score

**from** **sklearn.metrics** **import** average\_precision\_score

**from** **sklearn.metrics** **import** recall\_score

Implement the function *print\_evaluation\_scores* which calculates and prints to stdout:

* *accuracy*
* *F1-score macro/micro/weighted*
* *Precision macro/micro/weighted*

In [ ]:

**def** print\_evaluation\_scores(y\_val, predicted):

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

In [ ]:

print('Bag-of-words')

print\_evaluation\_scores(y\_val, y\_val\_predicted\_labels\_mybag)

print('Tfidf')

print\_evaluation\_scores(y\_val, y\_val\_predicted\_labels\_tfidf)

You might also want to plot some generalization of the [ROC curve](http://scikit-learn.org/stable/modules/model_evaluation.html#receiver-operating-characteristic-roc) for the case of multi-label classification. Provided function *roc\_auc* can make it for you. The input parameters of this function are:

* true labels
* decision functions scores
* number of classes

In [ ]:

**from** **metrics** **import** roc\_auc

%matplotlib inline

In [ ]:

n\_classes = len(tags\_counts)

roc\_auc(y\_val, y\_val\_predicted\_scores\_mybag, n\_classes)

In [ ]:

n\_classes = len(tags\_counts)

roc\_auc(y\_val, y\_val\_predicted\_scores\_tfidf, n\_classes)

**Task 4 (MultilabelClassification).** Once we have the evaluation set up, we suggest that you experiment a bit with training your classifiers. We will use *F1-score weighted* as an evaluation metric. Our recommendation:

* compare the quality of the bag-of-words and TF-IDF approaches and chose one of them.
* for the chosen one, try *L1* and *L2*-regularization techniques in Logistic Regression with different coefficients (e.g. C equal to 0.1, 1, 10, 100).

You also could try other improvements of the preprocessing / model, if you want.

In [ ]:

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

When you are happy with the quality, create predictions for *test* set, which you will submit to Coursera.

In [ ]:

test\_predictions = *######### YOUR CODE HERE #############*

test\_pred\_inversed = mlb.inverse\_transform(test\_predictions)

test\_predictions\_for\_submission = '**\n**'.join('**%i\t%s**' % (i, ','.join(row)) **for** i, row **in** enumerate(test\_pred\_inversed))

grader.submit\_tag('MultilabelClassification', test\_predictions\_for\_submission)

**Analysis of the most important features**

Finally, it is usually a good idea to look at the features (words or n-grams) that are used with the largest weigths in your logistic regression model.

Implement the function *print\_words\_for\_tag* to find them. Get back to sklearn documentation on [OneVsRestClassifier](http://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html) and [LogisticRegression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) if needed.

In [ ]:

**def** print\_words\_for\_tag(classifier, tag, tags\_classes, index\_to\_words, all\_words):

*"""*

*classifier: trained classifier*

*tag: particular tag*

*tags\_classes: a list of classes names from MultiLabelBinarizer*

*index\_to\_words: index\_to\_words transformation*

*all\_words: all words in the dictionary*

*return nothing, just print top 5 positive and top 5 negative words for current tag*

*"""*

print('Tag:**\t{}**'.format(tag))

*# Extract an estimator from the classifier for the given tag.*

*# Extract feature coefficients from the estimator.*

*######################################*

*######### YOUR CODE HERE #############*

*######################################*

top\_positive\_words = *# top-5 words sorted by the coefficiens.*

top\_negative\_words = *# bottom-5 words sorted by the coefficients.*

print('Top positive words:**\t{}**'.format(', '.join(top\_positive\_words)))

print('Top negative words:**\t{}\n**'.format(', '.join(top\_negative\_words)))

In [ ]:

print\_words\_for\_tag(classifier\_tfidf, 'c', mlb.classes, tfidf\_reversed\_vocab, ALL\_WORDS)

print\_words\_for\_tag(classifier\_tfidf, 'c++', mlb.classes, tfidf\_reversed\_vocab, ALL\_WORDS)

print\_words\_for\_tag(classifier\_tfidf, 'linux', mlb.classes, tfidf\_reversed\_vocab, ALL\_WORDS)