Text Classification with Python

Bastian Birkeneder

January 28, 2019

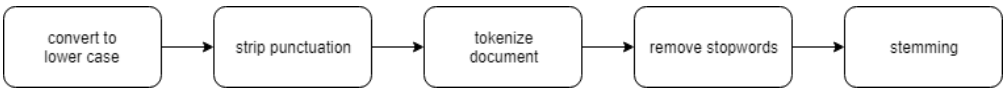
1 Introduction

This notebook provides a short introduction for classifying text corpora using the python packages scikit-learn and nltk. The Reuters corpus is used to demonstrate the easy use of these high level machine learning libraries.

In [1]:
```
%matplotlib notebook

import nltk
from nltk.corpus import reuters
import warnings
import numpy as np
```

In [2]:
```
# Prevents warnings during cross-validation
warnings.filterwarnings("ignore")

# Number of folds during cross-validation
k = 5

# Number of parallel computations (n_jobs parameter); -1 for utilizing the entire cpu
jobs = -1

# Pseudo-random number generator seed, for reproduceable results
seed = 42
```

In [3]:
```
## This code downloads the required packages.
## You can run `nltk.download('all')` to download everything.
#nltk.download('punkt')
nltk_packages = [
    ("reuters", "corpora/reuters.zip"),
    ("punkt", "tokenizers/punkt.zip")
]

for pid, fid in nltk_packages:
    try:
        nltk.data.find(fid)
    except LookupError:
        nltk.download(pid)
```



preprocessing

```
[nltk_data] Downloading package punkt to /home/bastian/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

1.1 Setting up train/test data

```
In [4]: X_train, y_train = zip(*[(reuters.raw(i), reuters.categories(i))
                                 for i in reuters.fileids()
                                 if i.startswith('training/')])
        X_test, y_test = zip(*[(reuters.raw(i), reuters.categories(i))
                               for i in reuters.fileids() if i.startswith('test/')])
```

```
In [5]: all_categories = sorted(list(set(reuters.categories())))
```

1.2 1. Preprocessing

A series of preprocessing steps will be applied before the classification step. First, each document in X_train (and X_test) will be converted to lower case and be stripped off all punctuations. Each document will be tokenized successively and for each token the stemmed form will be saved. A tf-idf vectorization will be used as word embedding. This frequency-based model can achieve good results for text classification tasks.[1]

To utilze the scikit-learn package (i.e. for using pipelines) the first two steps will be performed in a custom Transformer class.

```
In [6]: import string
        from sklearn.base import BaseEstimator, TransformerMixin


        class TMPreProcessor(BaseEstimator, TransformerMixin):
            def __init__(self):
                None

            def fit(self, X, y=None):
                return self

            def transform(self, X):
                processed = []

                # Convert to lower case and strip punctuation
                for i in range(len(X)):
                    text = X[i]
                    processed.append(text.lower().translate(
                        str.maketrans('', '', string.punctuation)))
                return processed
```

Since the used sklearn.feature_extraction.text.TfidfVectorizer class has a callable parameter tokenizer, the tokenization and stemming will be performed in a custom function. The parameter stop_words allows an easy removal of stopwords.

```
In [7]: from nltk import word_tokenize
        from nltk.stem.porter import PorterStemmer

        def tokenize(text):
            tokens = word_tokenize(text)
            stems = []
            stemmer = PorterStemmer()
            for token in tokens:
                stems.append(stemmer.stem(token))
            return stems
```

ngram_range also considers bi-grams. The max_features value is chosen arbitrary to reduce the feature dimensionality.

```
In [8]: from sklearn.feature_extraction.text import TfidfVectorizer

        preprocessor = TMPreProcessor()
        tfidf = TfidfVectorizer(
            tokenizer=tokenize,
            stop_words='english',
            ngram_range=(1, 2),
            max_features=5000)

        print('Transforming documents...')
        X_train = preprocessor.transform(X_train)
        X_train = tfidf.fit_transform(X_train)
        X_test = preprocessor.transform(X_test)
        X_test = tfidf.transform(X_test)
        print('Transformation finished!')

Transforming documents...
Transformation finished!
```

The labels/text categories (y_train and y_test) will be processed with the sklearn.preprocessing.MultiLabelBinarizer.

```
In [9]: from sklearn.preprocessing import MultiLabelBinarizer

        mlb = MultiLabelBinarizer()
        y_train = mlb.fit_transform(y_train)
        y_test = mlb.transform(y_test)
```

1.3 2. Finding an estimator

Several methods exist for the multi-class, multi-label classification; the baseline approach amounts to independently training one binary classifier for each label. The OVR (One-vs-the-rest/one-vs-all) strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes (out-of-class). In most cases, this is the default strategy for this kind of classification problem. The first step is to find an estimator suitable for the specific classification task. Therefore, the performance of a couple of baseline classifiers will be compared (i.e., using default parameters) based on their scored accuracy. A 5-fold cross-validation will be used to determine the mean accuracy of each classifier. The tested classifiers are: * SVM (linear kernel) * Naive Bayes * LogistitcRegression * k-nearest neighbors * AdaBoost classifier * Decission tree * Random forest

```
In [10]: from sklearn.svm import LinearSVC
         from sklearn.linear_model import LogisticRegression
         from sklearn.naive_bayes import BernoulliNB
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
         from sklearn.tree import DecisionTreeClassifier

         from sklearn.multiclass import OneVsRestClassifier

         from sklearn.model_selection import cross_val_score

         names = [
             "Linear SVM", "BernoulliNB", "LogisticRegression", "KNeighborsClassifier",
             "AdaBoostClassifier", "Random Forest", "Decision Tree"
         ]

         classifiers = [
             OneVsRestClassifier(LinearSVC(random_state=seed)),
             OneVsRestClassifier(BernoulliNB()),
             OneVsRestClassifier(
                 LogisticRegression(random_state=seed, solver='sag', max_iter=1000)),
             OneVsRestClassifier(KNeighborsClassifier()),
             OneVsRestClassifier(AdaBoostClassifier()),
             OneVsRestClassifier(RandomForestClassifier(random_state=seed)),
             OneVsRestClassifier(DecisionTreeClassifier(random_state=seed))
         ]

         print('Searching best estimator...')
         print()
         best_classifier = None
         for name, clf in zip(names, classifiers):
             scores = cross_val_score(clf, X_train, y_train, cv=k, n_jobs=jobs)
             print('Mean accuracy %s: %0.3f (+/- %0.3f)' % (name, scores.mean(),
                                                            scores.std() * 2))
             if not best_classifier:
                 best_classifier = (name, scores.mean())
             else:
```

```
                if best_classifier[1] < scores.mean():
                    best_classifier = (name, scores.mean())
        print()
        print('Best estimator: %s (mean acc %0.3f, %d-fold cross-validation)' %
              (best_classifier[0], best_classifier[1], k))
```

```
Searching best estimator...

Mean accuracy Linear SVM: 0.805 (+/- 0.030)
Mean accuracy BernoulliNB: 0.535 (+/- 0.045)
Mean accuracy LogisticRegression: 0.667 (+/- 0.032)
Mean accuracy KNeighborsClassifier: 0.401 (+/- 0.046)
Mean accuracy AdaBoostClassifier: 0.748 (+/- 0.011)
Mean accuracy Random Forest: 0.654 (+/- 0.029)
Mean accuracy Decision Tree: 0.686 (+/- 0.031)

Best estimator: Linear SVM (mean acc 0.805, 5-fold cross-validation)
```

The linear SVM yields the best results based on the accuracy. Since the accuracy of a classifier is not always the optimal performance measure, it can be advisable to further check other scoring metrics. In this case the F1 score (which is the weighted average of the precision and recall) is additionally computed to provide a better assessment of all classifiers.

```
In [11]: from sklearn.metrics import f1_score
         from sklearn.metrics import make_scorer

         print('Searching best estimator (F1 score) ...')
         print()
         best_classifier = None
         for name, clf in zip(names, classifiers):
             scores = cross_val_score(
                 clf,
                 X_train,
                 y_train,
                 cv=k,
                 n_jobs=jobs,
                 scoring=make_scorer(f1_score, average='micro'))
             print('Mean F1 score %s: %0.3f (+/- %0.3f)' % (name, scores.mean(),
                                                            scores.std() * 2))
             if not best_classifier:
                 best_classifier = (name, scores.mean())
             else:
                 if best_classifier[1] < scores.mean():
                     best_classifier = (name, scores.mean())
         print()
         print('Best estimator: %s (mean F1 score %0.3f, %d-fold cross-validation)' %
               (best_classifier[0], best_classifier[1], k))
```

```
Searching best estimator (F1 score) ...

Mean F1 score Linear SVM: 0.873 (+/- 0.023)
Mean F1 score BernoulliNB: 0.382 (+/- 0.007)
Mean F1 score LogisticRegression: 0.763 (+/- 0.030)
Mean F1 score KNeighborsClassifier: 0.513 (+/- 0.048)
Mean F1 score AdaBoostClassifier: 0.848 (+/- 0.011)
Mean F1 score Random Forest: 0.757 (+/- 0.022)
Mean F1 score Decision Tree: 0.814 (+/- 0.019)

Best estimator: Linear SVM (mean F1 score 0.873, 5-fold cross-validation)
```

The SVM also performs best in this case. As the next step one can try to tune the hyperparameter of the SVM, to optimize the model.

1.4 3. Tuning the model

There are a couple of different SVM implemetations in scikit-learn. Another noteworthy option is the sklearn.linear_model.SGDClassifier. It is a general linear classifier with stochastic gradient decent optimization. It gives a linear SVM with the loss='hinge' parameter.

```
In [12]: from sklearn.linear_model import SGDClassifier
         from sklearn.model_selection import cross_validate

         estimator = OneVsRestClassifier(
             SGDClassifier(random_state=seed, max_iter=1000, loss='hinge'))

         scoring = {'acc': 'accuracy', 'f1': make_scorer(f1_score, average='micro')}
         scores = cross_validate(
             estimator, X_train, y_train, cv=k, n_jobs=jobs, scoring=scoring)
         print('Mean accuracy %s: %0.3f (+/- %0.3f)' %
               ('Linear SVM (SGD)', scores['test_acc'].mean(),
                scores['test_acc'].std() * 2))
         print('Mean F1 score %s: %0.3f (+/- %0.3f)' % (
             'Linear SVM (SGD)', scores['test_f1'].mean(), scores['test_f1'].std() * 2))
```

```
Mean accuracy Linear SVM (SGD): 0.812 (+/- 0.036)
Mean F1 score Linear SVM (SGD): 0.878 (+/- 0.022)
```

The SGD optimization slightly increases both accuracy and F1 score, but comes at the cost of a higher computation time.

The alpha parameter of the SGDClassifier is the main hyperparameter which will be tuned. It multiplies the regularization term (defined by the penalty parameter) and is used to compute the learning_rate. To tune this parameter, the sklearn.model_selection.validation_curve will be used. A cross-validation for different alpha values will be performed to determine the best performance measure (F1 score). Note that the 'right' approach here would rather be an exhaustive grid search or random search with a combination of different parameters in order to find the optimal model.

However, the fine tuning process is a rather time and resource consuming procedure (and may include some extra steps like feature selection), hence the focus will be on just one parameter.

```
In [13]: from sklearn.model_selection import validation_curve, learning_curve
         from sklearn.multiclass import OneVsRestClassifier
         from sklearn.linear_model import SGDClassifier

         scoring = make_scorer(f1_score, average='micro')

         param_range = np.linspace(10**-6, 10**-3, 20)
         train_scores, test_scores = validation_curve(
             estimator,
             X_train,
             y_train,
             'estimator__alpha',
             param_range,
             cv=k,
             n_jobs=jobs,
             scoring=scoring)
```

Plotting the validation curve:

```
In [14]: import matplotlib.pyplot as plt

         train_scores_mean = np.mean(train_scores, axis=1)
         train_scores_std = np.std(train_scores, axis=1)
         test_scores_mean = np.mean(test_scores, axis=1)
         test_scores_std = np.std(test_scores, axis=1)

         plt.title("Validation Curve SVM")
         plt.xlabel("alpha")
         plt.ylabel("F1 Score")
         plt.ylim(0.0, 1.1)
         lw = 1
         plt.semilogx(
             param_range,
             train_scores_mean,
             label="Training score",
             color="darkorange",
             lw=lw)
         plt.fill_between(
             param_range,
             train_scores_mean - train_scores_std,
             train_scores_mean + train_scores_std,
             alpha=0.2,
             color="darkorange",
             lw=lw)
         plt.semilogx(
```

```
            param_range,
            test_scores_mean,
            label="Cross-validation score",
            color="navy",
            lw=lw)
        plt.fill_between(
            param_range,
            test_scores_mean - test_scores_std,
            test_scores_mean + test_scores_std,
            alpha=0.2,
            color="navy",
            lw=lw)
        plt.legend(loc="best")
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

One can observe, that the curve of the training score and cross-validation score converge for increasing alpha values. A high training score but a low cross-validation score is called overfitting. A low training score and cross-validation score indicates underfitting (in this case larger values for alpha). Normally, one tries to find a model which generalizes well for unknown data. The default parameter value 10^{-4} is already near the optimum.

Another aspect to evaluate a model is to cross-validate over various training sizes. This allows an assesment of the size of the training corpus, whether more training data would lead to better results.

In [15]:
```
from sklearn.model_selection import learning_curve

train_sizes = np.linspace(.1, 1.0, 5)
train_sizes, train_scores, test_scores = learning_curve(
    estimator,
    X_train,
    y_train,
    cv=k,
    n_jobs=jobs,
    train_sizes=train_sizes,
    scoring=scoring)
```

Plotting the learning curve:

In [16]:
```
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
```

```
        plt.figure()
        plt.title('Learning Curve SVM')
        plt.xlabel("Training examples")
        plt.ylabel("F1 Score")
        plt.ylim(0.0, 1.1)
        plt.grid()
        plt.fill_between(
            train_sizes,
            train_scores_mean - train_scores_std,
            train_scores_mean + train_scores_std,
            alpha=0.1,
            color="r")
        plt.fill_between(
            train_sizes,
            test_scores_mean - test_scores_std,
            test_scores_mean + test_scores_std,
            alpha=0.1,
            color="g")
        plt.plot(
            train_sizes,
            train_scores_mean,
            'o-',
            color="darkorange",
            label="Training score")
        plt.plot(
            train_sizes,
            test_scores_mean,
            'o-',
            color="navy",
            label="Cross-validation score")
        plt.legend(loc="best")
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

The cross-validation score (F1 Score) continuously increases with more training data and slowly converges with the training score. This indicates that more training data might improve the overall performance of the classifier.

A final cross-validation on the adjusted alpha:

```
In [17]: estimator = OneVsRestClassifier(
             SGDClassifier(
                 random_state=seed, max_iter=1000, loss='hinge', alpha=1.1 * (10**-4)))

         scoring = {'acc': 'accuracy', 'f1': make_scorer(f1_score, average='micro')}
```

```
        scores = cross_validate(
            estimator, X_train, y_train, cv=k, n_jobs=jobs, scoring=scoring)
        print('Mean accuracy %s: %0.3f (+/- %0.3f)' %
              ('Linear SVM (SGD)', scores['test_acc'].mean(),
               scores['test_acc'].std() * 2))
        print('Mean F1 score %s: %0.3f (+/- %0.3f)' % (
            'Linear SVM (SGD)', scores['test_f1'].mean(), scores['test_f1'].std() * 2))
```

```
Mean accuracy Linear SVM (SGD): 0.812 (+/- 0.035)
Mean F1 score Linear SVM (SGD): 0.878 (+/- 0.022)
```

Now the SVM can be trained with the whole training set. y_pred are the predicted categories of the test set.

```
In [18]: estimator.fit(X_train, y_train)
         y_pred = estimator.predict(X_test)
```

1.5 4. Evaluation

The overall performance of the classifier is evaluated using a number of different metrics. Apart from the accuracy, the F1 score, precision, and recall are measured (both micro-averaged and macro-averaged).

```
In [19]: from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
```

```
         def print_scores(y_test, y_pred):
             print('Accuracy: %0.3f' % accuracy_score(y_test, y_pred))
             print()
             print('F1 Score (micro-averaged): %0.3f' % f1_score(
                 y_test, y_pred, average='micro'))
             print('Precision (micro-averaged): %0.3f' % precision_score(
                 y_test, y_pred, average='micro'))
             print('Recall (micro-averaged): %0.3f' % recall_score(
                 y_test, y_pred, average='micro'))
             print()
             print('F1 Score (macro-averaged): %0.3f' % f1_score(
                 y_test, y_pred, average='macro'))
             print('Precision (macro-averaged): %0.3f' % precision_score(
                 y_test, y_pred, average='macro'))
             print('Recall (macro-averaged): %0.3f' % recall_score(
                 y_test, y_pred, average='macro'))
```

```
In [20]: print_scores(y_test, y_pred)
```

```
Accuracy: 0.820

F1 Score (micro-averaged): 0.872
```

```
Precision (micro-averaged): 0.946
Recall (micro-averaged): 0.809

F1 Score (macro-averaged): 0.453
Precision (macro-averaged): 0.598
Recall (macro-averaged): 0.391
```

The accuracy and F1 score of the unseen test set are similar to the cross-validated scores, which means the model minimizes the generalization error and thus ovefitting was indeed avoided. One can observe the significantly lower values of the macro-averaged scores. This can be further investigated with a classification report or confusion matrix.

```
In [21]: from sklearn.metrics import classification_report

         print(classification_report(y_test, y_pred, target_names=all_categories))
```

	precision	recall	f1-score	support
acq	0.98	0.95	0.97	719
alum	1.00	0.61	0.76	23
barley	1.00	0.64	0.78	14
bop	0.95	0.60	0.73	30
carcass	0.82	0.50	0.62	18
castor-oil	0.00	0.00	0.00	1
cocoa	1.00	0.94	0.97	18
coconut	1.00	0.50	0.67	2
coconut-oil	0.00	0.00	0.00	3
coffee	0.93	0.93	0.93	28
copper	1.00	0.89	0.94	18
copra-cake	0.00	0.00	0.00	1
corn	0.98	0.73	0.84	56
cotton	1.00	0.55	0.71	20
cotton-oil	0.00	0.00	0.00	2
cpi	1.00	0.46	0.63	28
cpu	0.00	0.00	0.00	1
crude	0.89	0.89	0.89	189
dfl	0.00	0.00	0.00	1
dlr	0.79	0.75	0.77	44
dmk	0.00	0.00	0.00	4
earn	0.99	0.98	0.98	1087
fuel	1.00	0.20	0.33	10
gas	1.00	0.41	0.58	17
gnp	0.97	0.86	0.91	35
gold	0.92	0.80	0.86	30
grain	0.98	0.84	0.90	149
groundnut	0.00	0.00	0.00	4
groundnut-oil	0.00	0.00	0.00	1

heat	1.00	0.60	0.75	5
hog	1.00	0.50	0.67	6
housing	1.00	0.50	0.67	4
income	0.00	0.00	0.00	7
instal-debt	0.00	0.00	0.00	1
interest	0.88	0.66	0.76	131
ipi	1.00	0.92	0.96	12
iron-steel	0.75	0.64	0.69	14
jet	0.00	0.00	0.00	1
jobs	1.00	0.52	0.69	21
l-cattle	0.00	0.00	0.00	2
lead	0.00	0.00	0.00	14
lei	1.00	1.00	1.00	3
lin-oil	0.00	0.00	0.00	1
livestock	0.83	0.62	0.71	24
lumber	1.00	0.17	0.29	6
meal-feed	1.00	0.11	0.19	19
money-fx	0.81	0.83	0.82	179
money-supply	0.93	0.74	0.82	34
naphtha	0.00	0.00	0.00	4
nat-gas	0.75	0.60	0.67	30
nickel	1.00	1.00	1.00	1
nkr	0.00	0.00	0.00	2
nzdlr	0.00	0.00	0.00	2
oat	0.00	0.00	0.00	6
oilseed	0.78	0.62	0.69	47
orange	1.00	0.82	0.90	11
palladium	0.00	0.00	0.00	1
palm-oil	1.00	0.60	0.75	10
palmkernel	0.00	0.00	0.00	1
pet-chem	0.00	0.00	0.00	12
platinum	1.00	0.29	0.44	7
potato	0.00	0.00	0.00	3
propane	0.00	0.00	0.00	3
rand	0.00	0.00	0.00	1
rape-oil	0.00	0.00	0.00	3
rapeseed	1.00	0.44	0.62	9
reserves	0.86	0.67	0.75	18
retail	1.00	0.50	0.67	2
rice	0.91	0.42	0.57	24
rubber	1.00	0.75	0.86	12
rye	0.00	0.00	0.00	1
ship	0.94	0.72	0.82	89
silver	1.00	0.12	0.22	8
sorghum	0.75	0.30	0.43	10
soy-meal	0.00	0.00	0.00	13
soy-oil	0.00	0.00	0.00	11
soybean	0.77	0.52	0.62	33

strategic-metal	0.00	0.00	0.00	11
sugar	0.93	0.75	0.83	36
sun-meal	0.00	0.00	0.00	1
sun-oil	0.00	0.00	0.00	2
sunseed	1.00	0.20	0.33	5
tea	0.00	0.00	0.00	4
tin	1.00	0.75	0.86	12
trade	0.86	0.76	0.81	117
veg-oil	1.00	0.41	0.58	37
wheat	0.93	0.73	0.82	71
wpi	1.00	0.60	0.75	10
yen	1.00	0.21	0.35	14
zinc	1.00	0.54	0.70	13
micro avg	0.95	0.81	0.87	3744
macro avg	0.60	0.39	0.45	3744
weighted avg	0.91	0.81	0.85	3744
samples avg	0.89	0.88	0.88	3744

Since all classes are weighted equally in the macro-averaged score calculation, classes with a smaller amount of sample data, which are harder to classify, can skew the overall macro-averaged score (e.g., the 'potato' category consists of only 3 documents, where no one was correctly classified).

1.6 5. Bonus: Neural Network

This section trains a neural network and needs additional python packages: * TensorFlow, Theano, or CNTK (as backend for keras) * keras * keras-tqdm (optional)

Since we have a multi-class, multi-label problem, the probabilites of each class should be independent of the other class probabilies. This can be achieved with the sigmoid activation function at the output layer of the neural network. All other layers will be computed using a rectifier function. Furthermore, each hidden layer will have a neuron dropout rate of 30% to prevent overfitting. Another important choice is the loss function. The binary_crossentropy loss is used instead of the categorical_crossentropy which is usually used in multi-class classification problems. This might seem unreasonable, but one wants to penalize each output node independently. The NN will be trained for 50 epochs and a batch size of 10 (number of samples per gradient update) using the ADADELTA optimization method.

In [23]:
```
import keras

from keras.models import Sequential
from keras.layers import Dense, Dropout

from keras_tqdm import TQDMNotebookCallback

model = Sequential()
```

```
        model.add(Dense(2000, input_dim=5000, activation='relu'))
        model.add(Dropout(.3))
        model.add(Dense(1000, activation='relu'))
        model.add(Dropout(.3))
        model.add(Dense(400, activation='relu'))
        model.add(Dropout(.3))
        model.add(Dense(len(all_categories), activation='sigmoid'))

        model.compile(
            loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
        model.fit(
            X_train,
            y_train,
            epochs=3,
            batch_size=10,
            verbose=0,
            callbacks=[TQDMNotebookCallback()])

        y_pred = (model.predict(X_test) >= .5) * 1
```

```
HBox(children=(IntProgress(value=0, description='Training', max=3, style=ProgressStyle(descripti
```

```
HBox(children=(IntProgress(value=0, description='Epoch 0', max=7769, style=ProgressStyle(descrip
```

```
HBox(children=(IntProgress(value=0, description='Epoch 1', max=7769, style=ProgressStyle(descrip
```

```
HBox(children=(IntProgress(value=0, description='Epoch 2', max=7769, style=ProgressStyle(descrip
```

```
In [24]: print_scores(y_test, y_pred)
```

```
Accuracy: 0.804

F1 Score (micro-averaged): 0.852
Precision (micro-averaged): 0.899
Recall (micro-averaged): 0.810

F1 Score (macro-averaged): 0.375
Precision (macro-averaged): 0.477
Recall (macro-averaged): 0.335
```

The NN performs similarly to the SVM.

1.7 References

Section **??** ZHANG, Wen; YOSHIDA, Taketoshi; TANG, Xijin. A comparative study of TF* IDF, LSI and multi-words for text classification. Expert Systems with Applications, 2011, 38. Jg., Nr. 3, S. 2758-2765.