School of Data and Computer Science, Sun Yat-sen University

SVM

18308045 谷正阳

May 16, 2021

Contents

1 Preprocessing

1.1 Feature extraction: TF-IDF

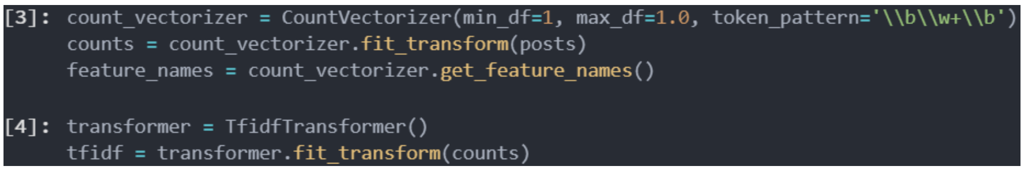
Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. The term-frequency helps to scale down the impact of very long documents, while the inverse document-frequency is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus. The formula that is used to compute the $tf - idf$ for a term t of a document d in a document set is

$$tf - idf(t, d) = tf(t, d)idf(t)$$

, and the idf is computed as

$$idf(t) = \log(\frac{n}{df(t) + 1})$$

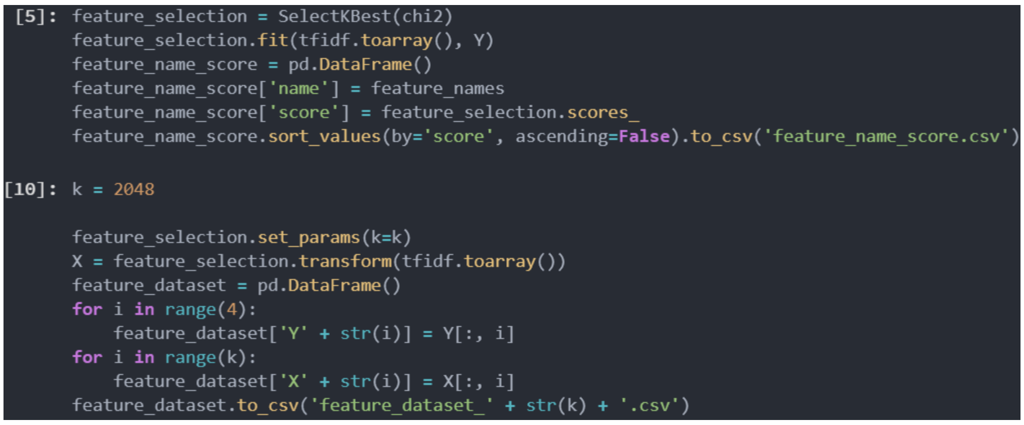
, where tf is the term-frequency in one classification, n is the total number of documents in the document set and $df(t)$ is the document frequency of t.



```
[3]: count_vectorizer = CountVectorizer(min_df=1, max_df=1.0, token_pattern='\\b\\w+\\b')
     counts = count_vectorizer.fit_transform(posts)
     feature_names = count_vectorizer.get_feature_names()

[4]: transformer = TfidfTransformer()
     tfidf = transformer.fit_transform(counts)
```

1.2 Feature selection: chi2

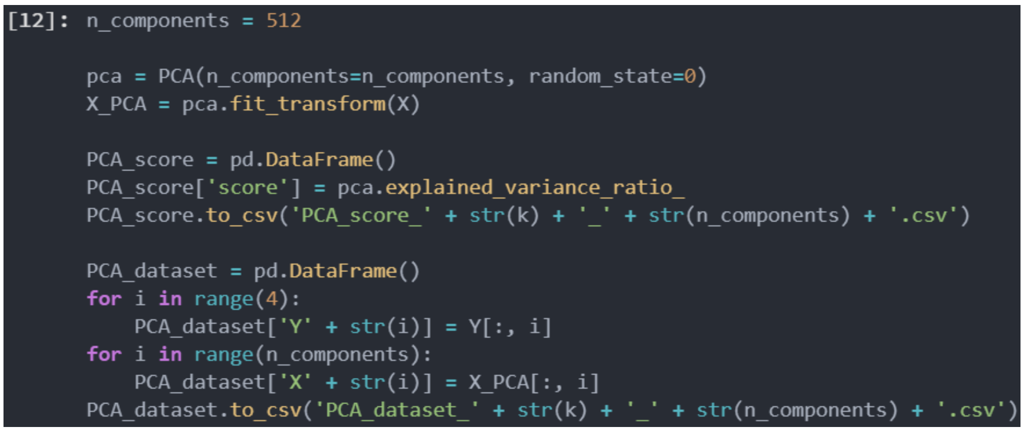
Since the number of features is too large, the time and space complexity can be very high and the lack of training set can lead to overfitting. Therefore, I use the chi2 score to select K best features.



```
 [5]: feature_selection = SelectKBest(chi2)
      feature_selection.fit(tfidf.toarray(), Y)
      feature_name_score = pd.DataFrame()
      feature_name_score['name'] = feature_names
      feature_name_score['score'] = feature_selection.scores_
      feature_name_score.sort_values(by='score', ascending=False).to_csv('feature_name_score.csv')

[10]: k = 2048

      feature_selection.set_params(k=k)
      X = feature_selection.transform(tfidf.toarray())
      feature_dataset = pd.DataFrame()
      for i in range(4):
          feature_dataset['Y' + str(i)] = Y[:, i]
      for i in range(k):
          feature_dataset['X' + str(i)] = X[:, i]
      feature_dataset.to_csv('feature_dataset_' + str(k) + '.csv')
```

1.3 Dimensionality reduction: PCA

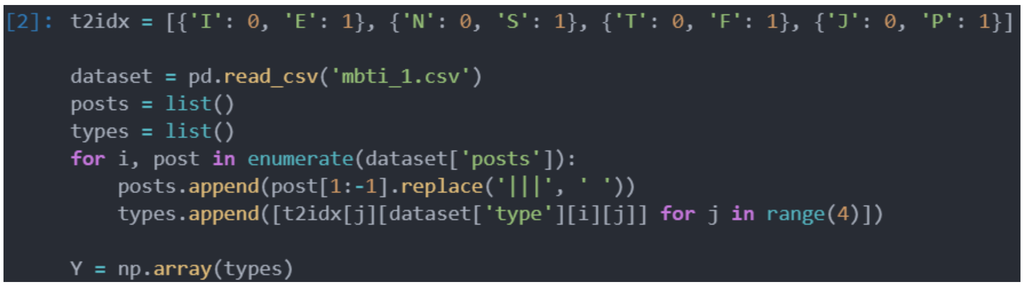
PCA is a way of linear dimensionality reduction, which uses Singular Value Decomposition of the data to project it to a lower dimensional space. The basic idea of PCA is to minimize the recounstruction error and meanwhile to ensure that the projections of all points are separated as far as possible.



```
[12]: n_components = 512

      pca = PCA(n_components=n_components, random_state=0)
      X_PCA = pca.fit_transform(X)

      PCA_score = pd.DataFrame()
      PCA_score['score'] = pca.explained_variance_ratio_
      PCA_score.to_csv('PCA_score_' + str(k) + '_' + str(n_components) + '.csv')

      PCA_dataset = pd.DataFrame()
      for i in range(4):
          PCA_dataset['Y' + str(i)] = Y[:, i]
      for i in range(n_components):
          PCA_dataset['X' + str(i)] = X_PCA[:, i]
      PCA_dataset.to_csv('PCA_dataset_' + str(k) + '_' + str(n_components) + '.csv')
```

1.4 Representation of outputs

The problem is to classify people into 16 personality types across 4 axis based on their posts. Therefore, a 4-output one-hot vector is a good representation of personality types.



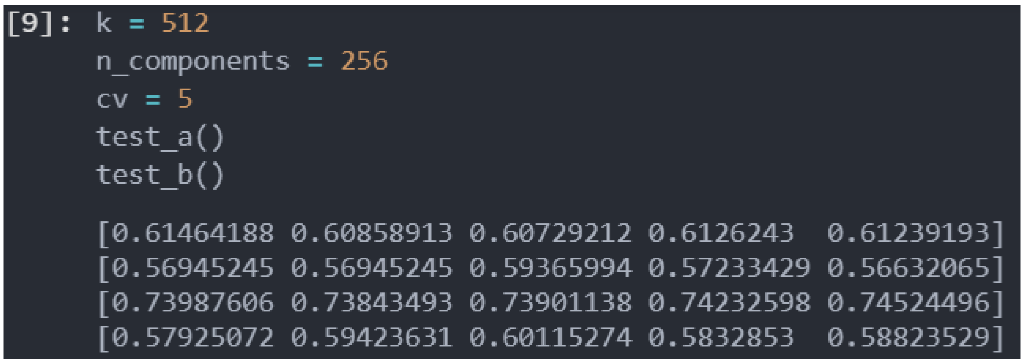
```
[2]: t2idx = [{'I': 0, 'E': 1}, {'N': 0, 'S': 1}, {'T': 0, 'F': 1}, {'J': 0, 'P': 1}]

     dataset = pd.read_csv('mbti_1.csv')
     posts = list()
     types = list()
     for i, post in enumerate(dataset['posts']):
         posts.append(post[1:-1].replace('|||', ' '))
         types.append([t2idx[j][dataset['type'][i][j]] for j in range(4)])

     Y = np.array(types)
```

2 Classification

I change the number of selected features, the number of features after PCA, and generate several csv files storing datasets and scores separately. Using these datasets, I test several hyperparameters' impact on the model.

2.1 The number of selected features

Firstly, I test the impact of different number of selected features.



```
[9]: k = 512
     n_components = 256
     cv = 5
     test_a()
     test_b()

     [0.61464188 0.60858913 0.60729212 0.6126243  0.61239193]
     [0.56945245 0.56945245 0.59365994 0.57233429 0.56632065]
     [0.73987606 0.73843493 0.73901138 0.74232598 0.74524496]
     [0.57925072 0.59423631 0.60115274 0.5832853  0.58823529]
```

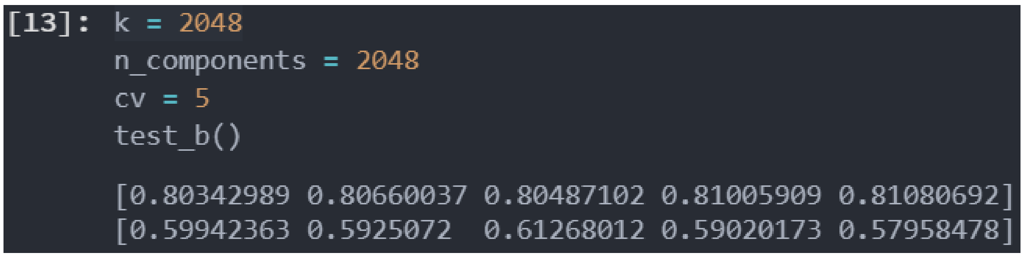
k stands for the number of selected features, n_components stands for the number of features after PCA, cv represents cross-validation splitter, and the four lines of outputs respectively represent training scores before PCA, testing scores before PCA, training scores after PCA and testing scores after PCA. The first 2 lines of outputs are what we should consider in this section.

k	Average training score	Average testing score
512	0.611107872	0.5742439559999999
1024	0.6302454880000001	0.577125136
2048	0.651083652	0.580121922
4096	0.6613730180000001	0.577124804
8192	0.688119534	0.57643296

When k is larger than 2048, overfitting apparently happens.

2.2 The number of features after PCA

Then I set the k as 2048 and test the impact of different number of features after PCA.



```
[13]: k = 2048
      n_components = 2048
      cv = 5
      test_b()

      [0.80342989 0.80660037 0.80487102 0.81005909 0.81080692]
      [0.59942363 0.5925072  0.61268012 0.59020173 0.57958478]
```

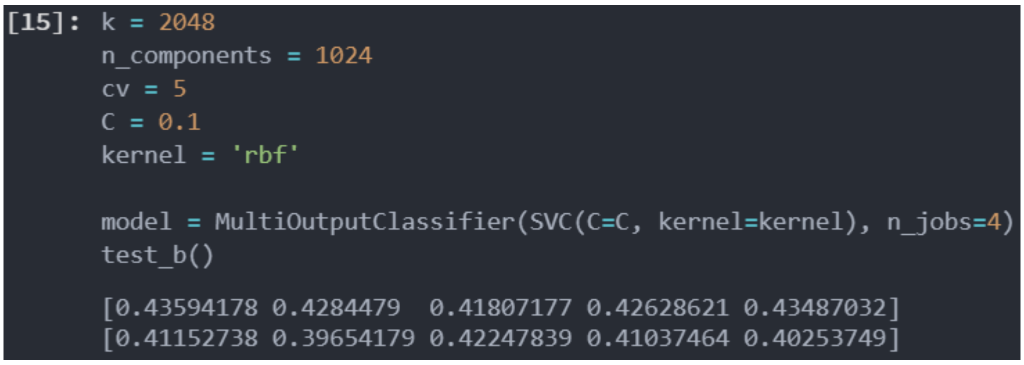
Except for the average training scores and average testing scroes, I also add the summation of percentage of variance which is stored in PCA_k_n_components_score.csv.

n_components	Average training score	Average testing score	Percentage of variance
512	0.7893992540000001	0.595341118	0.956029
1024	0.802570794	0.596262778	0.993766
2048	0.8071534579999999	0.5948794920000001	1

It is shown that 1024 is the best n_components.

2.3 Regularization parameter for l2 penalty

There is also a regularization parameter in SVM implemented by sklearn, so I test it's impact on the model.



```
[15]: k = 2048
      n_components = 1024
      cv = 5
      C = 0.1
      kernel = 'rbf'

      model = MultiOutputClassifier(SVC(C=C, kernel=kernel), n_jobs=4)
      test_b()

      [0.43594178 0.4284479  0.41807177 0.42628621 0.43487032]
      [0.41152738 0.39654179 0.42247839 0.41037464 0.40253749]
```

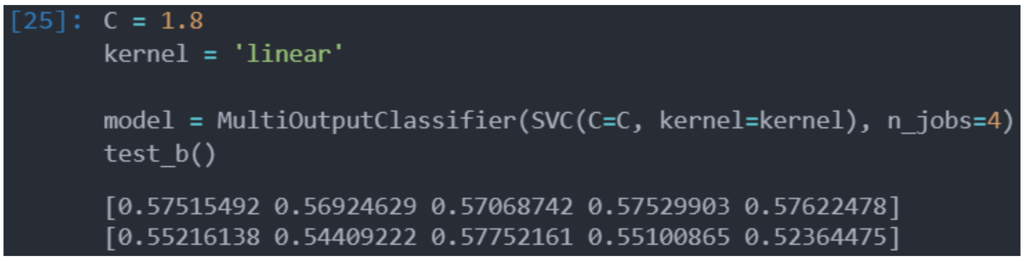
C represents the regularization parameter and kernel stands for kernel function used in SVM model.

C	Average training score	Average testing score
0.1	0.42872359600000004	0.4086919379999999
1	0.802570794	0.596262778
1.5	0.860819626	0.6032962700000001
1.7	0.876585084	0.605140584
1.8	0.8846551619999999	0.6052557920000001
1.9	0.89180294	0.605140454
2	0.899181334	0.60444881
2.5	0.925812676	0.603065392
3	0.944056904	0.6015663659999999
5	0.982072868	0.596263774
10	0.998443634	0.5880782040000001

It is shown that 1.8 is the best C.

2.4 Kernel function

Finally I test the impact of kernel function on the model.



```
[25]: C = 1.8
      kernel = 'linear'

      model = MultiOutputClassifier(SVC(C=C, kernel=kernel), n_jobs=4)
      test_b()

      [0.57515492 0.56924629 0.57068742 0.57529903 0.57622478]
      [0.55216138 0.54409222 0.57752161 0.55100865 0.52364475]
```

kernel	Average training score	Average testing score
linear	0.573322488	0.549685722
poly	0.785191332	0.45134602399999996
rbf	0.8846551619999999	0.6052557920000001
sigmoid	0.368255656	0.34966464999999997

It is shown that rbf is the best kernel.

3 Conclusions

There are some aspects that can be improved. For instance, the CountVectorizer can filter out words that appear frequently or rarely by setting max_df and min_df. And kernel functions can be also used in PCA. Additionally, there are lots of other hyperparameters of SVM which I just use the default value in this experiment.