Tagging Project pub

January 21, 2021

0.1 Natural Language Processing for Twitter User Descriptions

Natural language processing (NLP) helps computers understand, interpret and manipulate human languages such as English. This notebook will walk through building a binary classifier to predict if the Twitter user is a gamer or a programmer using that user's profile as input.

[1]:
```
import numpy as np
import pandas as pd
pd.set_option('max_colwidth', 5000)
```

0.2 Input data

The previous notebook streamed Twitter data to a MySQL database using keywords [python, c++, java, pokemongo, animalcrossing, acnh]. The first three keywords capture users that are more likely to be programmers, and the last three keywords are my favorite games that capture users more likely to be gamers. The streaming lasts around 80 hours, and there were more programmers than gamers who post tweets within the same time. Therefore, I ran extra 10 hours for keywords of gamers to make the data balance. Finally, I collected 154,683 unique tweets from 63,019 individual users. To ensure data purity, two additional conditions are applied to clean the raw samples:

```
1. The user's profile is not empty
2. The user's tweets contain label information
```

After filtering, the total number of users reduce to 63,015.

[2]:
```
from sqlalchemy import create_engine
import pymysql
sqlEngine = create_engine('mysql+pymysql://root:sqlserverpasscode@127.0.0.1',␣
 ↪pool_recycle=3600)

dbConnection = sqlEngine.connect()

df_full = pd.read_sql("SELECT twitterid, userid, description, tweet FROM␣
 ↪Twitterdb.Tag WHERE tweet LIKE '%python%' OR tweet LIKE '%java%' OR tweet LIKE␣
 ↪'%c++%'  OR tweet LIKE '%PokemonGo%' OR tweet LIKE '%AnimalCrossing%' OR tweet␣
 ↪LIKE '%ACNH%'", dbConnection);
pd.set_option('display.expand_frame_repr', False)
dbConnection.close()
```

```
print('number of users: ', len(df_full.userid.unique()))
print('number of twitters',len(df_full.twitterid.unique()))
df_full.head()
```

```
number of users:  63019
number of twitters 154683
```

[2]:

```
              twitterid               userid
description
tweet
0  1350256705272074242  1261351467404849152
None                                                        RT @celebrity_studs:
Ross has a one-eyed python between his legs https://t.co/kXvoNgDWL9
1  1350256723207081984           2319287015
None                                                            @animalcrossing
Can we add Celeste as a permanent character? https://t.co/65XOCe3XCr
2  1350256724305997824            182033390              Geek, desenvolvedor e
sla oq escrever aqui     RT @RDS150: You deserve someone that will pog at you
like Celeste pogs at the starry sky \n\n#AnimalCrossing #ACNH
https://t.co/vMsUy9K6x6
3  1350256740294664194  1337908138620702720  Programmer | Python Dev | Front-End
Dev | Python Tutor
RT @elonmusk: Monty Python is amazing\nhttps://t.co/UJq94IWT88
4  1350256740584054784  1228668163715411968            Hello. I'm a bot.
@Camillekinoti created me  RT @wruthrum131: Day #22 of #100DaysOfCode \n\n-
Worked through @freeCodeCamp Scientific Coding with #Python, now at 43% \n\nWas
pretty busy...
```

[3]:

```
import warnings
warnings.filterwarnings("ignore")
from re import search

#The user's profile is not empty
df = df_full[['userid','description','tweet']].dropna()
#if user posted more than one tweets, only keep one of them
df = df.drop_duplicates(subset=['userid']).reset_index(drop = True)

# user tagging
df['tweet'] = df['tweet'].str.lower()
programmers = ['python','c++','java','sql']
gamers = ['pokemongo','animalcrossing', 'acnh']

df['type'] = pd.Series(index = df.index)
for i in range(df.shape[0]):
    if any(x in df['tweet'][i] for x in programmers):
        df['type'][i] = 'programmer'
    if any(x in df['tweet'][i] for x in gamers):
```

```
        df['type'][i] = 'gamer'
    if any(x in df['tweet'][i] for x in programmers) and any(x in df['tweet'][i]␣
 ↪for x in gamers):
        df['type'][i] = 'both'

#only keep gamers and programmers
df = df[(df.type == 'gamer')|(df.type == 'programmer')]
print('number of users: ', df.shape[0])
print('user distribution: ')
print(df.type.value_counts())

df.head()
```

```
number of users:  63015
user distribution:
programmer    36834
gamer         26181
Name: type, dtype: int64
```

[3]:

```
                userid                                        description
 tweet        type
 0  1261351467404849152                                              None
 rt @celebrity_studs: ross has a one-eyed python between his legs
 https://t.co/kxvongdwl9  programmer
 1           2319287015                                              None
 @animalcrossing can we add celeste as a permanent character?
 https://t.co/65xoce3xcr       gamer
 2            182033390             Geek, desenvolvedor e sla oq escrever aqui
 rt @rds150: you deserve someone that will pog at you like celeste pogs at the
 starry sky \n\n#animalcrossing #acnh https://t.co/vmsuy9k6x6       gamer
 3  1337908138620702720  Programmer | Python Dev | Front-End Dev | Python Tutor
 rt @elonmusk: monty python is amazing\nhttps://t.co/ujq94iwt88  programmer
 4  1228668163715411968            Hello. I'm a bot. @Camillekinoti created me
 rt @wruthrum131: day #22 of #100daysofcode \n\n- worked through @freecodecamp
 scientific coding with #python, now at 43% \n\nwas pretty busy...  programmer
```

[4]:

```
df.type = [0 if type == "programmer" else 1 for type in df.type]
print(df.type.value_counts())

df = df[df.description != 'None']
print(df.type.value_counts())
```

```
0    36834
1    26181
Name: type, dtype: int64
0    32174
1    24483
```

```
Name: type, dtype: int64
```

0.3 Data preprocessing

Computers can only handle binary bytes, and the ultimate goal of text recognition is to convert the text data into numerical data and train computers to understand the underlying information. NLP does precisely this.

0.3.1 Tokenization

Tokenization means to split the document into individual words, but it still needs more cleaning, and the steps are following:

```
1. Lowercase all letters in the sentence
2. Only keep alphabetic words
3. Lemmatize all tokens (such as days into day)
4. Tokenize the sentence (split sentence into words)
5. Remove stop words (such as: and, the)
```

[5]:
```
import nltk
from nltk.corpus import stopwords
import re
swords = stopwords.words("english")
#words = set(nltk.corpus.words.words())

from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.stem import PorterStemmer
porter=PorterStemmer()
def stemSentence(sentence):
    token_words=word_tokenize(sentence)
    token_words
    stem_sentence=[]
    for word in token_words:
        stem_sentence.append(porter.stem(word))
        stem_sentence.append(" ")
    return "".join(stem_sentence)


text = df.description[20]
print(text)

text = text.lower()
print('')
print('lower')
print(text)

text = re.sub('[^a-z]', ' ', str(text))
print('')
```

```python
print('keep a-z')
print(text)

text = stemSentence(text)
print('')
print('lemma')
print(text)

text = nltk.word_tokenize(text)
print('')
print('token')
print(text)

text = [word for word in text if word not in swords]
print('')
print('stop words')
print(text)

#data = [word for word in data if word in words]
#print('')
#print('english words')
#print(data)

text = ' '.join(text)
print('')
print('join')
print(text)
```

```
Not very smart but still alive

lower
not very smart but still alive

keep a-z
not very smart but still alive

lemma
not veri smart but still aliv

token
['not', 'veri', 'smart', 'but', 'still', 'aliv']

stop words
['veri', 'smart', 'still', 'aliv']

join
veri smart still aliv
```

[6]:
```python
def token(text):
    text = text.lower()
    text = re.sub('[^a-z]', ' ', str(text))
    text = stemSentence(text)
    text = nltk.word_tokenize(text)
    text = [word for word in text if word not in swords]
    text = ' '.join(text)
    return text

#apply token function on every description
df['description'] = df['description'].apply(lambda x: token(x))

# remove records with empty description
#(for description only has stop words, it will become empty after removing stop␣
 ↪words)
df = df[(df.description != '')].reset_index(drop = True)
```

0.3.2 CountVectorizer

Data Tokenization transform sentence into a bag of words and CountVectorizer will transform all the words into numbers.

Take the example:

```
text1: ['I like pokemongo']
text2: ['I like python']
```

There are four unique tokens [I, like, pokemongo, python] in the entire dataset, which means the indices are in the range of 0 to 4. Then user's description could be represented as a vector with four elements. If a component is in the description file, 1s will be put to the corresponding indices. Otherwise, 0s will be put to the index when the description file does not contain those elements.

Therefore, texts will change to

```
text1: [1, 1, 1, 0]
text2: [1, 1, 0, 1]
```

However, in practice, the number of unique tokens would be huge, and due to the memory limits, it is not possible to store all the vectors. And the majority of vectors are usually composed of zeros, and only a small portion of them is ones, which is also called a sparse vector. Large-sized sparse vectors are usually stored in sparse representations where only the indices and the corresponding non-zero values are stored. For example,

```
Given a sparse vector [0, 0, 0, 1, 0, 2, 0, 0], the sparse representations of it is
(0, 3) 1
(0, 5) 2
```

Which means at location row 0, with column 3 and 5 have values 1 and 2 respectively.

The CountVectorizer() function from sklearn will handle all the steps.

0.3.3 TfidfTransformer

While CountVectorizer works fine in most cases, it is better if Tfidftransformer is applied next to CountVectorizer. Tf stands for term frequency, it is the normalized form of CountVectorizer, and tf-idf is calculated by term-frequency times the inverse of document frequency. The formula is

tfidf(t,d)=tf(t,d)idf(t), where idf(t) = log[n/df(t)]+1

From the formula, when the document frequency of term t is approaching the number of entire document set n, the logarithmic would be approaching 0. And the weight for the term tt would go towards 1. In contrast, the term with a lower document frequency would have a higher idf weight to magnify its term-frequency tf(t, d)tf(t,d).

The goal of Tfidf is to scale down the impact of tokens that occur in the majority of the document set.

The TfidfTransformer() in sklearn will perform the function. Note: TfidfVectorizer().fit_transform(corpus) = TfidfTransformer().fit_transform(CountVectorizer().fit_transform(corpus))

0.4 Classification

0.4.1 Train test split

[7]:
```
from sklearn.model_selection import train_test_split
X = df.description
y = df.type
X_train, X_test, y_train, y_test = train_test_split(df.description, df.type,␣
 ↪test_size = 0.3)

print('==> total number of users: ', df.shape[0])
print(' ')
print('==> training dataset: ', X_train.shape[0])
print(' ')
print('==> testing dataset: ', X_test.shape[0])
print(' ')
print('==> train label:')
print( y_train.value_counts())
print(' ')
print('==> test label:')
print(y_test.value_counts())
```

```
==> total number of users:  54312

==> training dataset:  38018

==> testing dataset:  16294

==> train label:
0    21625
1    16393
```

```
Name: type, dtype: int64

==> test label:
0    9276
1    7018
Name: type, dtype: int64
```

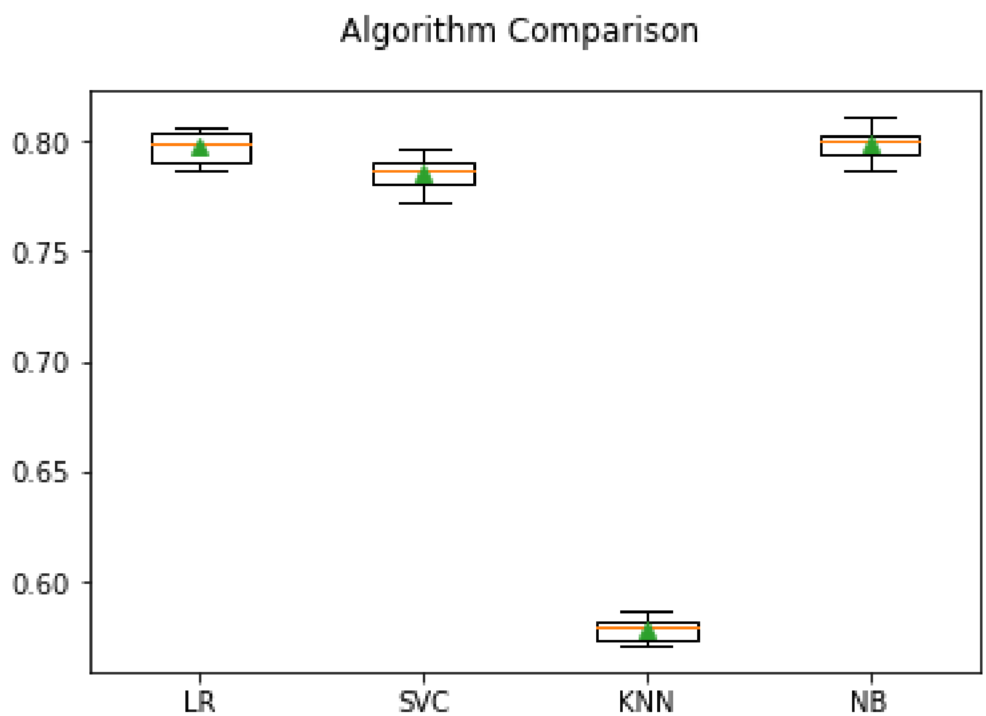
0.4.2 Model selection

[8]:

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

from sklearn import model_selection

from sklearn.linear_model import LogisticRegression
#from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
#from sklearn.tree import DecisionTreeClassifier

import matplotlib.pyplot as plt

models = []
models.append(('LR', LogisticRegression))
#models.append(('RF', RandomForestClassifier))#slow
models.append(('SVC', LinearSVC))
models.append(('KNN', KNeighborsClassifier))
#models.append(('DT', DecisionTreeClassifier)) #slow
models.append(('NB', MultinomialNB))

results = []
names = []
for name, model in models:

    Classifier = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('model', model()),
    ])

    kfold = model_selection.KFold(n_splits=10, random_state=7)
    cv_results = model_selection.cross_val_score(Classifier, X_train, y_train,␣
 ↪cv=kfold)
    results.append(cv_results)
    names.append(name)
```

```
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)

fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results,showmeans=True)
ax.set_xticklabels(names)
plt.show()
```

```
LR: 0.797070 (0.007305)
SVC: 0.785733 (0.007005)
KNN: 0.578463 (0.005315)
NB: 0.798885 (0.006701)
```



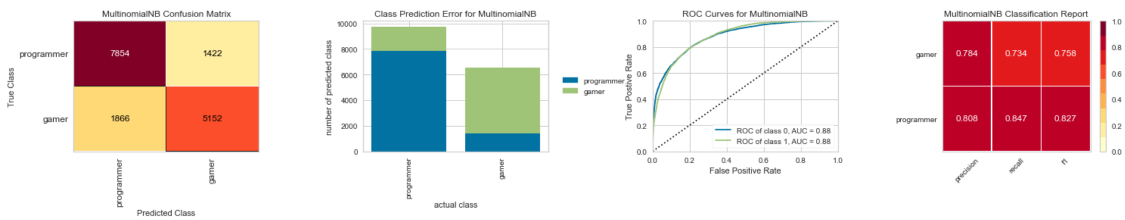
The training time for tree-based classifiers Decision Tree and Random Forest is too slow to be considered. Therefore, the baseline models only include Logistic regression (LR), K Nearest Neighbors (KNN), Support Vector Classifier (SVC), and Naive Bayes (NB). After running 10 fold cross-validation, LR, SVC, and NB showed better performance than KNN. The next step is hyperparameter tuning on these three models.

0.4.3 Baseline models performance on test dataset

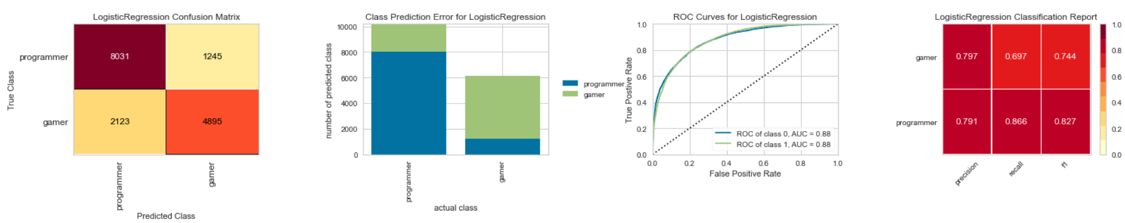
[9]:
```
# performance visualization tools
from yellowbrick.classifier import ClassPredictionError
from yellowbrick.classifier import ClassificationReport
from yellowbrick.classifier import ROCAUC
from yellowbrick.classifier import ConfusionMatrix

def metric_plot(model):
    fig, axes = plt.subplots(1, 4, figsize=(20,4))

    visualgrid = [
        ClassPredictionError(model, ax=axes[1], classes = ['programmer',␣
 ↪'gamer']),
        ClassificationReport(model, ax=axes[3], classes = ['programmer',␣
 ↪'gamer']),
        ROCAUC(model, micro=False, macro=False, ax=axes[2]),
        ConfusionMatrix(model, ax=axes[0], classes = ['programmer', 'gamer']),
    ]

    for viz in visualgrid:
        viz.fit(X_train, y_train)
        viz.score(X_test, y_test)
        viz.finalize()
    plt.show()

def svc_plot(model):
    fig, axes = plt.subplots(1, 4, figsize=(20,4))

    visualgrid = [
        ClassPredictionError(model, ax=axes[1], classes = ['programmer',␣
 ↪'gamer']),
        ClassificationReport(model, ax=axes[3], classes = ['programmer',␣
 ↪'gamer']),
        ROCAUC(model, micro=False, macro=False, per_class = False, ax=axes[2]),
        ConfusionMatrix(model, ax=axes[0], classes = ['programmer', 'gamer']),
    ]

    for viz in visualgrid:
        viz.fit(X_train, y_train)
        viz.score(X_test, y_test)
        viz.finalize()
    plt.show()

def model_pipline(classifier):
    model = Pipeline([
        ('vect', CountVectorizer()),
```

```python
        ('tfidf', TfidfTransformer()),
        ('model', classifier()),
        ])
    return model

## plot model performance
for m in [MultinomialNB, LogisticRegression]:
    print('model: ', m)
    metric_plot(model_pipline(m))
    print(' ')

print('model: LinearSVC')
svc_plot(model_pipline(LinearSVC))
```

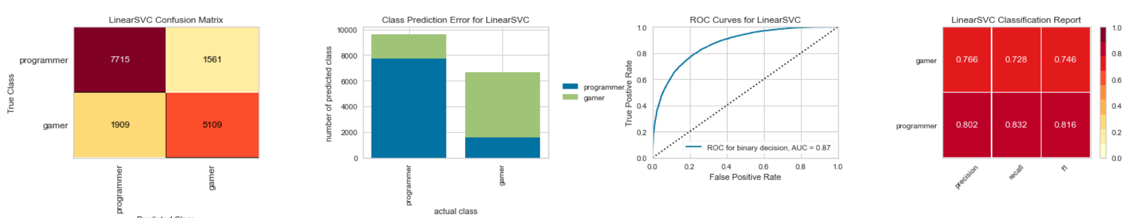
```
model:  &lt;class 'sklearn.naive_bayes.MultinomialNB'&gt;
```




```
model:  &lt;class 'sklearn.linear_model._logistic.LogisticRegression'&gt;
```




```
model: LinearSVC
```



0.4.4 GridSearchCV – Multinomial Naive Bayes

GridSearchCV is a submodule provided by scikit-learn, which combines grid search and cross-validation into one entity. It explicitly searches over a specified set of parameters. For each combination of parameters, it performs k-fold cross-validation automatically.

Some basic parameters are predefined to reduce the search time when initializing the pipeline, such as ngram_range for CountVectorizer and use_idf for TfidfTransformer. ngram_range stands for the range of n-gram to be extracted. Here I choose only to consider 1-gram and 2-grams sequences.

Two parameters for two modules in the pipeline are explicitly set in the parameter grid:

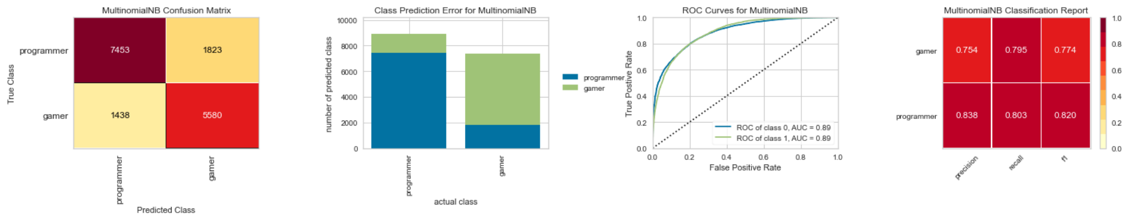
```
max_df (maximal document-frequency) of CountVectorizer: all the tokens that have a document-freq

alpha of Naive Bayes classifier: the smoothing coefficient that counts for unseen feature points
```

[10]:
```
from sklearn.model_selection import GridSearchCV


model = Pipeline([
    ('vect', CountVectorizer(ngram_range=(1, 2))),
    ('tfidf', TfidfTransformer(use_idf = True)),
    ('nb', MultinomialNB(fit_prior = False))
])

parameters = {
    'vect__max_df': ( 0.1, 0.3, 0.5),
    'nb__alpha': np.linspace(0.4, 1, 7)
}

nb_search = GridSearchCV(model, parameters, cv=10, n_jobs=12, verbose=2)
nb_search.fit(X_train, y_train)


print("Best Score: ", nb_search.best_score_)
print("Best Params: ", nb_search.best_params_)
```

```
Fitting 10 folds for each of 21 candidates, totalling 210 fits

[Parallel(n_jobs=12)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=12)]: Done  17 tasks      | elapsed:    7.6s
[Parallel(n_jobs=12)]: Done 138 tasks      | elapsed:   37.6s
[Parallel(n_jobs=12)]: Done 210 out of 210 | elapsed:   54.5s finished

Best Score:  0.8023566640800663
Best Params:  {'nb__alpha': 0.5, 'vect__max_df': 0.1}
```

Naive Bayes gridsearch model performance Compares to the default model, the best model after hyperparameter tuning has better performance: the f1 score for class gamer has increased from 0.758 to 0.774, the f1 score for class programmer has decreased from 0.827 to 0.820, and the AUC has risen from 0.88 to 0.89.

```
[11]: metric_plot(nb_search.best_estimator_)
```



0.4.5 GridSearchCV – Logistic Regression

To reduce the search time, besides the basic parameters mentioned above, I also set max_df as 0.1, which is from GridSearchSV results from SVC.

Three parameters for Logistic regression are explicitly set in the parameter grid:

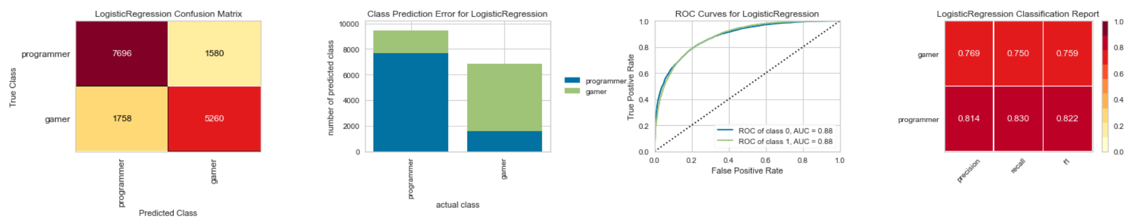
```
penalty: Regularization is a way to avoid overfitting by penalizing high-valued regression coeff

L1, also known as the Lasso penalty, is equal to the absolute value of coefficients' magnitude.
L2, also known as Ridge penalty, equals the square of the magnitude of coefficients. The same fa

C: Inverse of regularization strength; must be a positive float. Smaller values specify stronger

max_iter: Maximum number of iterations taken for the solvers to converge.
```

```
[12]: lr_pipeline = Pipeline([
          ('vect', CountVectorizer(ngram_range=(1, 2), max_df = 0.1)),
          ('tfidf', TfidfTransformer(use_idf = True)),
          ('lr', LogisticRegression())
      ])

      grid_params = {
          'lr__penalty': ['l1', 'l2'],
          'lr__C': [0.1, 1, 5],
          'lr__max_iter': [50, 100, 200],
      }

      lr_search= GridSearchCV(lr_pipeline, grid_params , cv=10, n_jobs=12, verbose=2)
      lr_search.fit(X_train, y_train)
      print("Best Score: ", lr_search.best_score_)
      print("Best Params: ", lr_search.best_params_)
```

```
Fitting 10 folds for each of 18 candidates, totalling 180 fits

[Parallel(n_jobs=12)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=12)]: Done  17 tasks      | elapsed:   10.8s
[Parallel(n_jobs=12)]: Done 138 tasks      | elapsed:  1.6min
[Parallel(n_jobs=12)]: Done 180 out of 180 | elapsed:  2.5min finished

Best Score:  0.796333366132919
Best Params:  {'lr__C': 5, 'lr__max_iter': 100, 'lr__penalty': 'l2'}
```

Logistic regression gridsearch model performance Compares to the default model, the best model after hyperparameter tuning has similar performance: the f1 score for class gamer has increased from 0.744 to 0.759, the f1 score for class programmer has decreased from 0.827 to 0.822, and the AUC stayed same as 0.88

[13]:
```python
metric_plot(lr_search.best_estimator_)
```



0.4.6 GridSearchCV – Support Vector Classifier

The GridSearch methods for SVC and Logistic regression are exactly the same.

[14]:
```python
svc_pipeline = Pipeline([
    ('vect', CountVectorizer(ngram_range=(1, 2), max_df = 0.1)),
    ('tfidf', TfidfTransformer(use_idf = True)),
    ('svc', LinearSVC())
])

grid_params = {
    'svc__penalty': ['l1', 'l2'],
    'svc__C': [0.1, 1, 5],
    'svc__max_iter': [50, 100, 200],
}
svc_search = GridSearchCV(svc_pipeline, grid_params , cv=10, n_jobs=12,␣
 ↪verbose=2)
svc_search.fit(X_train, y_train)
print("Best Score: ", svc_search.best_score_)
print("Best Params: ", svc_search.best_params_)
```

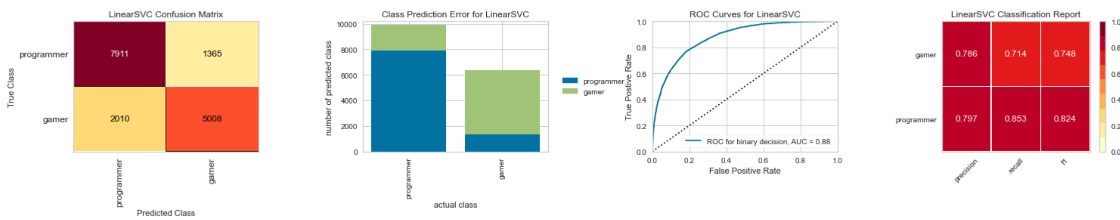
```
Fitting 10 folds for each of 18 candidates, totalling 180 fits
```

```
[Parallel(n_jobs=12)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=12)]: Done  17 tasks      | elapsed:    6.7s
[Parallel(n_jobs=12)]: Done 138 tasks      | elapsed:   42.3s
[Parallel(n_jobs=12)]: Done 180 out of 180 | elapsed:  1.0min finished
```

```
Best Score:  0.7964387676711229
Best Params:  {'svc__C': 0.1, 'svc__max_iter': 50, 'svc__penalty': 'l2'}
```

Support Vector Classifier gridsearch model performance Compares to the default model, the best model after hyperparameter tuning has better performance: the f1 score for class gamer has increased from 0.746 to 0.748, the f1 score for class programmer has decreased from 0.816 to 0.824, and the AUC stayed same as 0.88

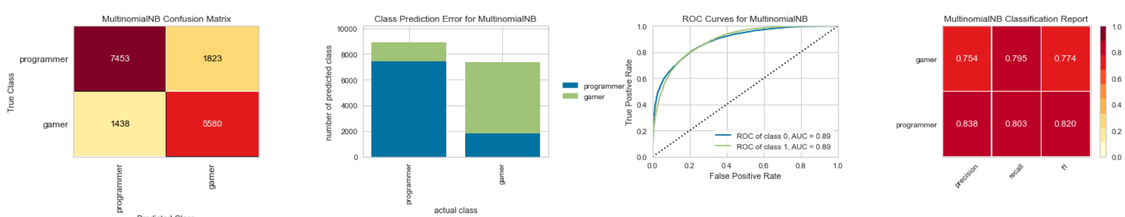
[15]:
```
svc_plot(svc_search.best_estimator_)
```



0.5 Conclusion

After comparing all the models, Multinomial Naive Bayes classifier has the best performace.

[18]:
```
final_model = nb_search.best_estimator_
metric_plot(final_model)
```



[39]:
```
import pickle
pickle.dump(final_model, open('mnb_model.pkl', 'wb'))
```

[ ]: