Drug-Target Interaction Prediction

In [1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split as tts

from sklearn.metrics import precision_recall_curve
```

Data Wrangling

In this section we would read and show the data

In [2]:
```
PATH = 'D:/Fall 2021/SYSC5405 Pattern Classification and Experiment Design/Project/' #

# read the data
df = pd.read_csv(PATH + 'train_data.csv')
df.head()
```

Out[2]:

	G26	G26_Target Sequence_in_SMILES_perc	G26_SMILES_in_Target Sequence_perc	G26_ARRO	G26_SMILES_base	G26_SMIL
0	5.073946	0.954991	0.062554	16.739510	5.522422	
1	6.826617	0.001154	0.014306	60569.338400	5.590014	
2	5.177106	0.818811	0.930755	1.312142	5.556566	
3	5.179001	0.183497	0.970300	5.616494	5.258327	
4	5.822746	0.029429	0.326915	103.942684	5.621003	

5 rows × 338 columns

In [3]:
```
print('Data size:', df.shape)
```

```
Data size: (109479, 338)
```

Simple EDA

- In this section we will choose some linked features and apply **simple Exploratory data analysis (EDA)**

In [4]:
```
# select only features which start with G26
g26_features = [col for col in df.columns if 'G26' in col]
```

```
# show some features
g26_features
```

Out[4]:
```
['G26',
 'G26_Target Sequence_in_SMILES_perc',
 'G26_SMILES_in_Target Sequence_perc',
 'G26_ARRO',
 'G26_SMILES_base',
 'G26_SMILES_base_perc',
 'G26_Target Sequence_base',
 'G26_Target Sequence_base_perc',
 'G26_fdp_SMILES_base',
 'G26_fdp_Target Sequence_base',
 'G26_fd_SMILES_base',
 'G26_fd_Target Sequence_base',
 'G26_std_SMILES_dist',
 'G26_std_Target Sequence_dist']
```

In [5]:
```
# create a dataset for speacial features
g26_df = df[g26_features]
print('New dataset contains:', len(g26_df.columns), 'features')
```

```
New dataset contains: 14 features
```

In [6]:
```
# show some rows
g26_df.head()
```

Out[6]:

	G26	G26_Target Sequence_in_SMILES_perc	G26_SMILES_in_Target Sequence_perc	G26_ARRO	G26_SMILES_base	G26_SMIL
0	5.073946	0.954991	0.062554	16.739510	5.522422	
1	6.826617	0.001154	0.014306	60569.338400	5.590014	
2	5.177106	0.818811	0.930755	1.312142	5.556566	
3	5.179001	0.183497	0.970300	5.616494	5.258327	
4	5.822746	0.029429	0.326915	103.942684	5.621003	

In [7]:
```
# show some statistics
g26_df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75
G26	109479.0	5.371929	0.461602	4.772719	5.067627	5.182776	5.5108
G26_Target Sequence_in_SMILES_perc	109479.0	0.534610	0.396936	0.000577	0.077323	0.646278	0.9267
G26_SMILES_in_Target Sequence_perc	109479.0	0.646215	0.348931	0.000054	0.327703	0.800479	0.9568
G26_ARRO	109479.0	5689.440099	169694.853922	1.000054	1.235749	3.246470	32.0879
G26_SMILES_base	109479.0	5.412270	0.260594	5.046103	5.223091	5.377441	5.5804

	count	mean	std	min	25%	50%	75
G26_SMILES_base_perc	109479.0	0.111702	0.010476	0.073860	0.104443	0.111368	0.1177
G26_Target Sequence_base	109479.0	5.670389	0.416734	5.032211	5.351949	5.624845	5.8969
G26_Target Sequence_base_perc	109479.0	0.131994	0.051213	0.043897	0.102100	0.118146	0.1422
G26_fdp_SMILES_base	109479.0	-0.422908	0.397982	-0.918638	-0.815349	-0.536065	0.0357
G26_fdp_Target Sequence_base	109479.0	-0.514221	0.357255	-0.955886	-0.828982	-0.660248	-0.1888
G26_fd_SMILES_base	109479.0	-0.006519	0.083325	-0.359096	-0.054075	-0.022496	0.0095
G26_fd_Target Sequence_base	109479.0	-0.050406	0.076102	-0.298188	-0.095845	-0.053042	-0.0168
G26_std_SMILES_dist	109479.0	0.322601	1.972052	-4.272021	-0.803506	-0.257326	0.8180
G26_std_Target Sequence_dist	109479.0	-0.303092	1.692982	-5.109258	-1.204099	-0.618807	0.1451

Data Pre-Processing

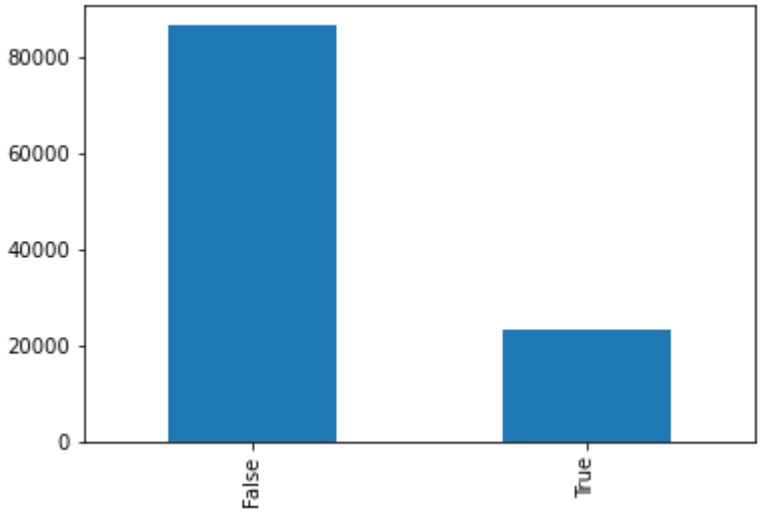
In [8]:
```
# check for nan values (features wise)
nadf = df.isna().sum()
nadf
```

Out[8]:
```
G26                                  0
G26_Target Sequence_in_SMILES_perc   0
G26_SMILES_in_Target Sequence_perc   0
G26_ARRO                             0
G26_SMILES_base                      0
                                    ..
G10_fd_Target Sequence_base          0
G10_std_SMILES_dist                  0
G10_std_Target Sequence_dist         0
KIBA                                 0
Label                                0
Length: 338, dtype: int64
```

In [9]:
```
# check for nan values (rows wise)
nadf.sum()
```

Out[9]:
```
0
```

In [10]:
```
# show labels balance
df.Label.value_counts().plot(kind='bar');
```



In [11]:
```
print(f'True Kiba proportion {df.Label.mean()*100:.2f}%.')
```

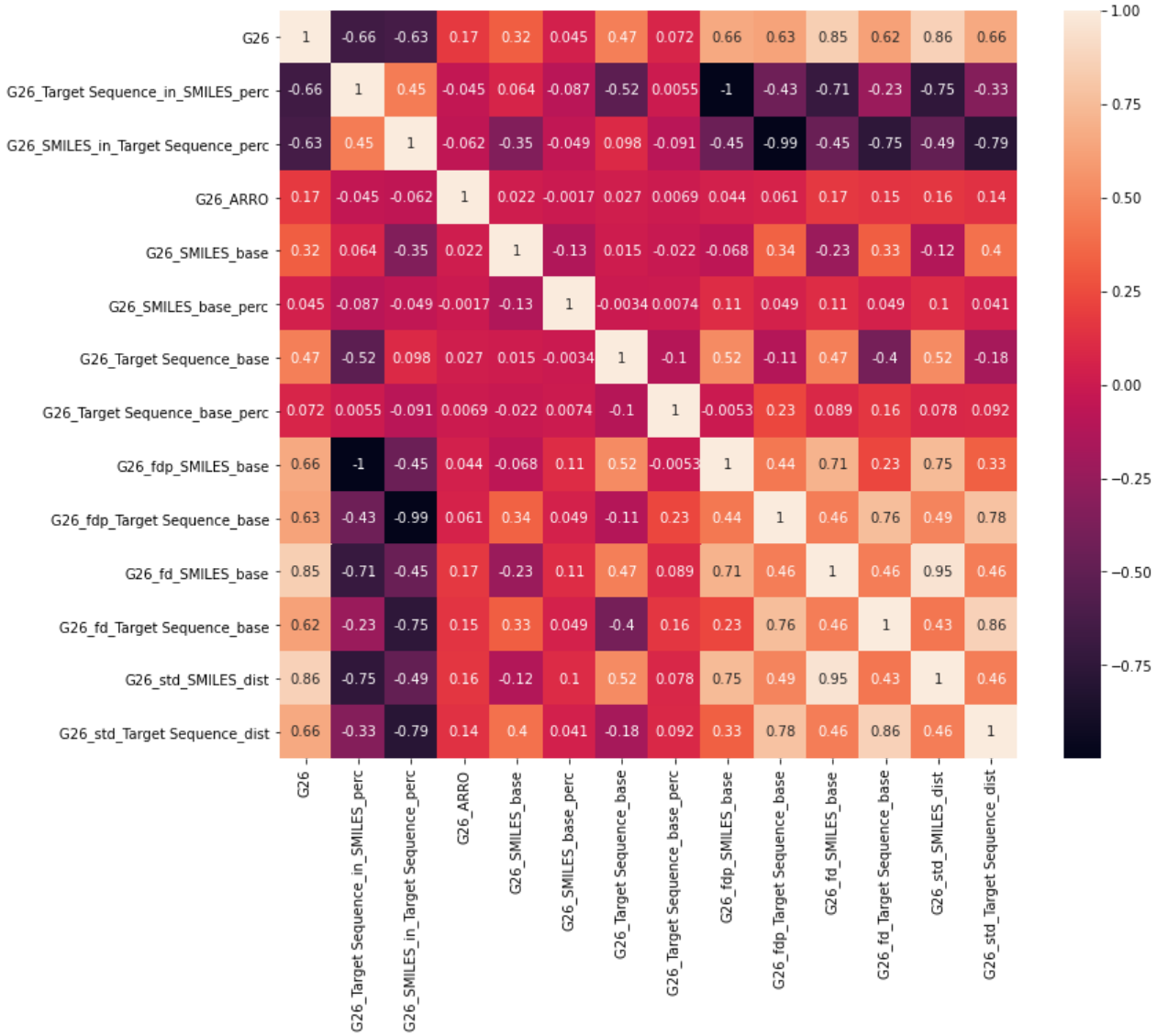
```
True Kiba proportion 21.15%.
```

In [12]:
```
# convert labels to integers
df['Label'] = df['Label'].astype('int')
df['Label'][:5]
```

Out[12]:
```
0    0
1    0
2    0
3    0
4    0
Name: Label, dtype: int32
```

Data Visualization

In [13]:
```
# show G26 feature correlation
corr = g26_df.corr()

# show correlation map
plt.figure(figsize=(12, 10))
sns.heatmap(corr, annot=True, fmt='.2g', );
```



Data Split

In [14]:
```
# split the data into train and test,
# by closer to the proportion of the actual test set  (8178 rows)

train, test = tts(df, test_size=0.08, random_state=111, stratify=df['Label'])

train.shape, test.shape
```

Out[14]:
```
((100720, 338), (8759, 338))
```

In [15]:
```
# show testset label counts
y_test = test['Label']

y_test.value_counts()
```

Out[15]:
```
0    6906
1    1853
Name: Label, dtype: int64
```

Data Normalize

- **Standard Scaler**, to normalize the feature and get a wider PCA range.

In [16]:
```
# Standard Scaler
from sklearn.preprocessing import StandardScaler

# define scaler
scaler = StandardScaler()

# extract features
features = train.iloc[:, :-2]
features_test = test.iloc[:, :-2]

# scale the features we captured
features_scaled = scaler.fit_transform(features.values)
features_scaled_test = scaler.transform(features_test.values)

# store it in dataframe
df_ = pd.DataFrame(features_scaled, columns=features.columns)
df_test = pd.DataFrame(features_scaled_test, columns=features_test.columns)

df_.shape , df_test.shape
```

Out[16]:
```
((100720, 336), (8759, 336))
```

Features Extractions

we will apply 2 method for extracts new features:

1. PCA: we will apply pca for each linked features.
2. KMeans: we will apply kmeans cluster to extract new 7 features.

In [17]:
```
# pca
from sklearn.decomposition import PCA

pca = PCA(n_components=2, random_state=111)

pca_df = pd.DataFrame()
pca_df_test = pd.DataFrame()

g_ids = []

for i in range(50):
    G_cols = [col for col in df.columns if f'G{i}' in col]
    if G_cols:
        g_df = df_[G_cols]
        g_df_test = df_test[G_cols]

        g_ids.append(i)

        pca_values = pca.fit_transform(g_df)
        pca_values_test = pca.transform(g_df_test)

        pca_df[f'G{i}_pca_{1}'] = pca_values[:, 0]
```

```
        pca_df[f'G{i}_pca_{2}'] = pca_values[:, 1]

        pca_df_test[f'G{i}_pca_{1}'] = pca_values_test[:, 0]
        pca_df_test[f'G{i}_pca_{2}'] = pca_values_test[:, 1]
```

In [18]:
```
pca_df.shape, pca_df_test.shape
```

Out[18]: ((100720, 50), (8759, 50))

In [19]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=7, random_state=111, verbose=0).fit(df_.values)

X_cd = kmeans.transform(df_.values)
X_cd_test = kmeans.transform(df_test.values)

cent_df = pd.DataFrame(X_cd, columns=[f"Centroid_{i}" for i in range(X_cd.shape[1])])
cent_df_test = pd.DataFrame(X_cd_test, columns=[f"Centroid_{i}" for i in range(X_cd_tes
```

In [20]:
```
# concatenate all features with scaled values
X = pd.concat([df_, pca_df, cent_df], axis=1)
X_test = pd.concat([df_test, pca_df_test, cent_df_test], axis=1)

y = train['Label']

X.shape, y.shape, X_test.shape
```

Out[20]: ((100720, 393), (100720,), (8759, 393))

Features Selections

we tried apply boosting important features for selecting important features, but we realized that it takes much time and does not affect the KNN performance.

In [21]:
```
#from xgboost import XGBClassifier
#from xgboost import plot_importance

## fit model no training data
#model = XGBClassifier(random_state=111, n_jobs=-1).fit(X_, y_)


# feature_important = model.get_booster().get_score(importance_type='cover')
# feature_important = dict(sorted(feature_important.items(), key=lambda item: item[1],

# keys = list(feature_important.keys())
# values = list(feature_important.values())

#SF = (values > np.mean(values)).sum() # cover > mean
#selected_features = keys[:SF]

# # plot feature importance
# plot_importance(model, max_num_features=25, importance_type='cover',)
# plt.show()
```

In [22]:
```
# we will choose all features
X = X[:] #X_[:selected_features]
X_test = X_test[:] #X_test[:selected_features]

X.shape, X_test.shape
```

Out[22]: ((100720, 393), (8759, 393))

Balancing Classes

In [23]:
```
from imblearn.over_sampling import RandomOverSampler

# oversample classes from 21% to 40%
oversample = RandomOverSampler(random_state=111, sampling_strategy=0.40, shrinkage=0.01

X_, y_ = oversample.fit_resample(X, y)

np.unique(y_, return_counts=True)
```

Out[23]: (array([0, 1]), array([79418, 31767], dtype=int64))

In [24]:
```
# np.save('train_data_clean.npy', X_)
# np.save('train_labels_clean.npy', y_)
```

Establish Experiment

In [25]:
```
# models imports
from sklearn.neighbors import KNeighborsClassifier #as KNN
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

from sklearn.model_selection import GridSearchCV

from sklearn.pipeline import Pipeline
```

In [26]:
```
# helper function to visualize our performances
def plot_precision_recall_curve(y_true, probs):
    precision, recall, thresholds = precision_recall_curve(y_true, probs)

    prerecall50 = precision[recall >= 0.5]
    #prerecall50 = precision
    max_precision = prerecall50.max()
    mean_precision = prerecall50.mean()
    std_precision = prerecall50.std()

    #prerecall50 = precision[recall >= 0.5]
    prerecall501 = precision
    #max_precision1 = prerecall501.max()
    mean_precision1 = prerecall501.mean()
    std_precision1 = prerecall501.std()
```

```
    idx = np.where(precision == max_precision)[0][0]
    prerecall_pair = (recall[idx], precision[idx])
    best_th = thresholds[idx]

    fig = plt.figure(figsize=(12,6))
    plt.step(recall, precision, color='r', alpha=0.2, where='post')
    plt.fill_between(recall, precision, step='post', alpha=0.2, color='#F59B00')
    plt.scatter(prerecall_pair[0], prerecall_pair[1], marker = 'x',
                label=f'precison-recall pair at\nTH:{best_th:.3f}\nPrecision:{max_preci

    plt.xlabel('Recall', fontsize=14)
    plt.ylabel('Precision', fontsize=14)
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])
    plt.title('Precision-Recall curve: \n Maximum Precision value (for Recall >= 50%) =
              max_precision), fontsize=16)

    plt.legend()
    plt.show()

    return best_th, mean_precision, std_precision, mean_precision1, std_precision1
```

In [27]:

```
# helper function to visualize our performances
def plot_cm_matrix(y_valid, y_pred):
    # Compute and plot the Confusion matrix
    cf_matrix = confusion_matrix(y_valid, y_pred)

    categories  = ['False Class','True Class']
    group_names = ['True Neg','False Pos', 'False Neg','True Pos']
    group_values = [f'{value}' for value in cf_matrix.flatten()]

    labels = [f'{v1}\n{v2}' for v1, v2 in zip(group_names, group_values)]
    labels = np.asarray(labels).reshape(2,2)

    #plt.figure(figsize=(6, 5))
    sns.heatmap(cf_matrix, annot = labels, cmap = 'Blues',fmt = '',
                xticklabels = categories, yticklabels = categories)

    plt.xlabel("Predicted values", fontdict = {'size':14}, labelpad = 10)
    plt.ylabel("Actual values"   , fontdict = {'size':14}, labelpad = 10)
    plt.title ("Confusion Matrix", fontdict = {'size':18}, pad = 20)

    plt.show()

    print('\n', classification_report(y_valid, y_pred))
```

Train classifier

- We used **StratifiedKFold (CV)** to split the train data and predict the probability for each class and store it in arrays, then we take the mean probabilities (around axis 0) values for 5 splits.

In [28]:

```
from sklearn.model_selection import StratifiedKFold
```

```
N_SPLIT = 5
skf = StratifiedKFold(n_splits=N_SPLIT, shuffle=True, random_state=111)

valid_preds = []
test_preds = []

for fold, (trn_idx, val_idx) in enumerate(skf.split(X_, y_), 1):
    print(f'Fold-{fold} Training...')
    X_train, X_valid = X_.iloc[trn_idx], X_.iloc[val_idx]
    y_train, y_valid = y_.iloc[trn_idx], y_.iloc[val_idx]

    clf = KNeighborsClassifier(n_neighbors=9, weights='distance', algorithm='brute')

    knn = Pipeline([
        ('ss', StandardScaler()),
        ('knn', clf),
    ])

    knn.fit(X_train, y_train)

    y_pred_train = knn.predict(X_train)
    #plot_cm_matrix(y_train, y_pred_train) # train cm

    print('Training Done. Starting Validate..')
    y_pred_ = knn.predict_proba(X_valid)
    valid_preds.append(y_pred_)
    y_pred = y_pred_.argmax(1)

    print('Validation accuracy:', (y_pred == y_valid).mean())
    #plot_cm_matrix(y_valid, y_pred) # valid cm

    print('Testing...')
    y_pred_test = knn.predict_proba(X_test)
    test_preds.append(y_pred_test)
    y_pred = y_pred_test.argmax(1)
    #plot_cm_matrix(y_test, y_pred) # test cm per split

    print('Testing accuracy', (y_pred == y_test).mean())
    print('-'  * 60)
```

```
Fold-1 Training...
Training Done. Starting Validate..
Validation accuracy: 0.8694967846382156
Testing...
Testing accuracy 0.8349126612627013
------------------------------------------------------------
Fold-2 Training...
Training Done. Starting Validate..
Validation accuracy: 0.8707559472950488
Testing...
Testing accuracy 0.8349126612627013
------------------------------------------------------------
Fold-3 Training...
Training Done. Starting Validate..
Validation accuracy: 0.8676979808427395
Testing...
Testing accuracy 0.8326292955816874
------------------------------------------------------------
Fold-4 Training...
Training Done. Starting Validate..
Validation accuracy: 0.8691370238791204
```

```
Testing...
Testing accuracy 0.8351409978308026
------------------------------------------------------------
Fold-5 Training...
Training Done. Starting Validate..
Validation accuracy: 0.8680577416018348
Testing...
Testing accuracy 0.835369334398904
------------------------------------------------------------
```

Testing & expected accuracy

PLOT Precision-Recall curve of our model's perormance.

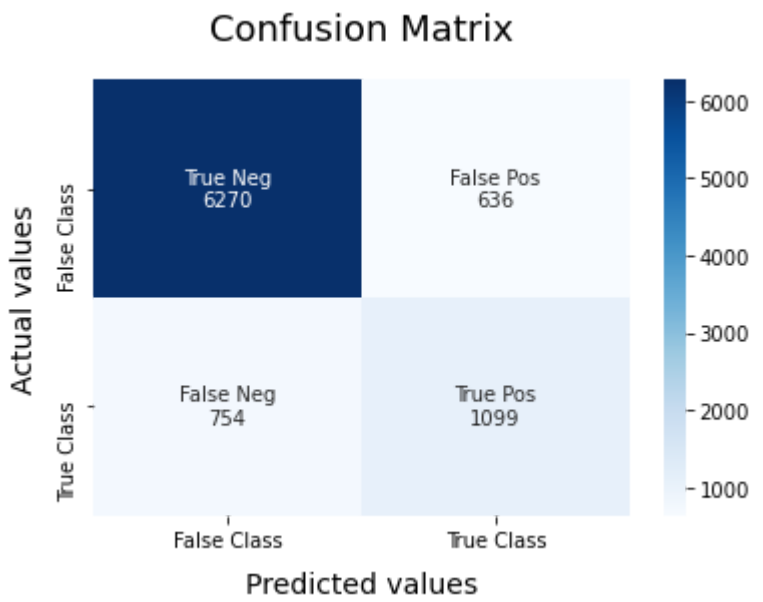
In [52]:
```
# predict the propablities
y_pred_ = np.mean(test_preds, axis=0)
y_pred = y_pred_.argmax(axis=1)

# show  precision recall curve for testset
best_th, mean_precision, std_precision = plot_precision_recall_curve(y_test, y_pred_[:,
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-52-a767e1f8d623> in <module>
      4
      5 # show  precision recall curve for testset
----> 6 best_th, mean_precision, std_precision = plot_precision_recall_curve(y_test, y_
pred_[:, 1])

<ipython-input-48-363daa070bc7> in plot_precision_recall_curve(y_true, probs)
     12     idx = np.where(precision == max_precision)[0][0]
     13     prerecall_pair = (recall[idx], precision[idx])
---> 14     best_th = thresholds[idx]
     15
     16     fig = plt.figure(figsize=(12,6))

IndexError: index 6637 is out of bounds for axis 0 with size 6637
```

In [53]:
```
#print('Best Threshold=%f, G-Mean=%.3f, G-Std=%.3f' % (best_th, mean_precision, std_pre
print('Best Threshold=%f, G-Mean=%.3f, G-Std=%.3f' % (best_th, mean_precision1, std_pre
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-53-2adaf90b5378> in <module>
----> 1 print('Best Threshold=%f, G-Mean=%.3f, G-Std=%.3f' % (best_th, mean_precision,
 std_precision))

NameError: name 'best_th' is not defined
```

In [54]:
```
# test confusion matrix
plot_cm_matrix(y_test, y_pred)
```



	precision	recall	f1-score	support
0	0.89	0.91	0.90	6906
1	0.63	0.59	0.61	1853
accuracy			0.84	8759
macro avg	0.76	0.75	0.76	8759
weighted avg	0.84	0.84	0.84	8759

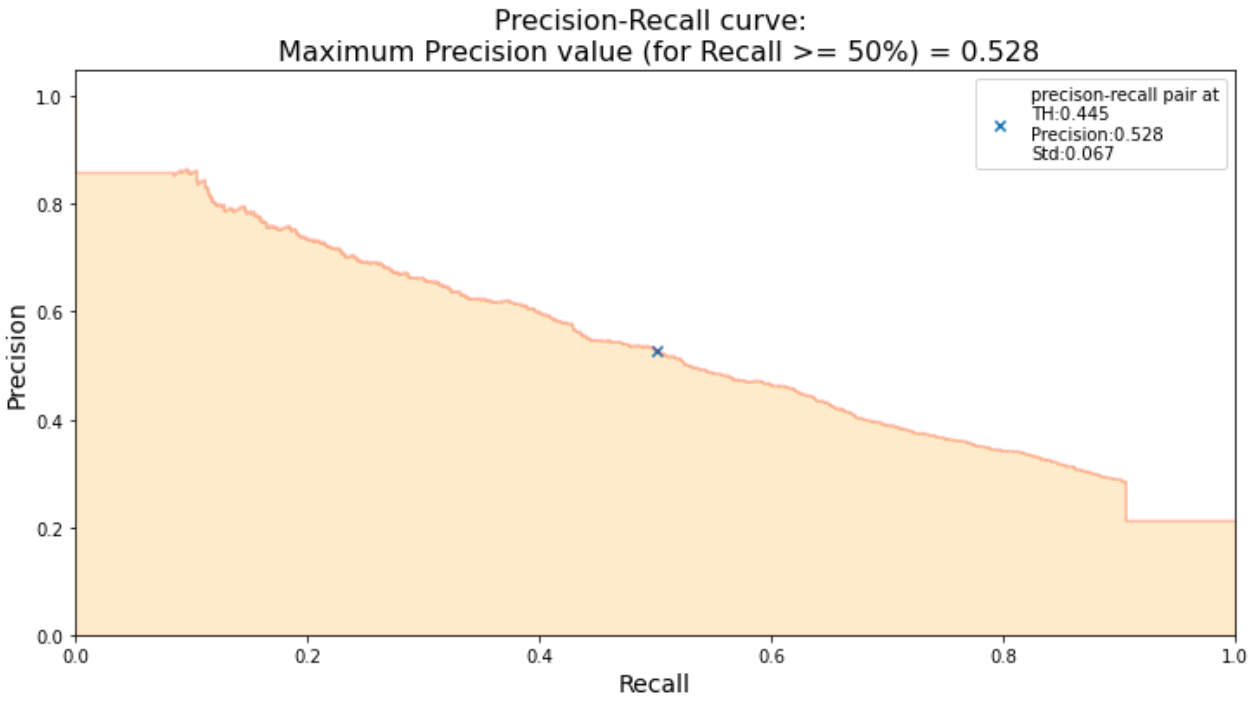
Meta-learning approaches (bagging)

In [33]:
```
from sklearn.ensemble import BaggingClassifier

bagging_clf = BaggingClassifier(base_estimator=knn, n_estimators=5, random_state=111, v

bagging_clf.fit(X_, y_)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   15.3s finished
```

Out[33]:
```
BaggingClassifier(base_estimator=Pipeline(steps=[('ss', StandardScaler()),
                                                 ('knn',
                                                  KNeighborsClassifier(algorithm='brut
e',
                                                                       n_neighbors=9,
                                                                       weights='distanc
e'))]),
                  n_estimators=5, random_state=111, verbose=1)
```

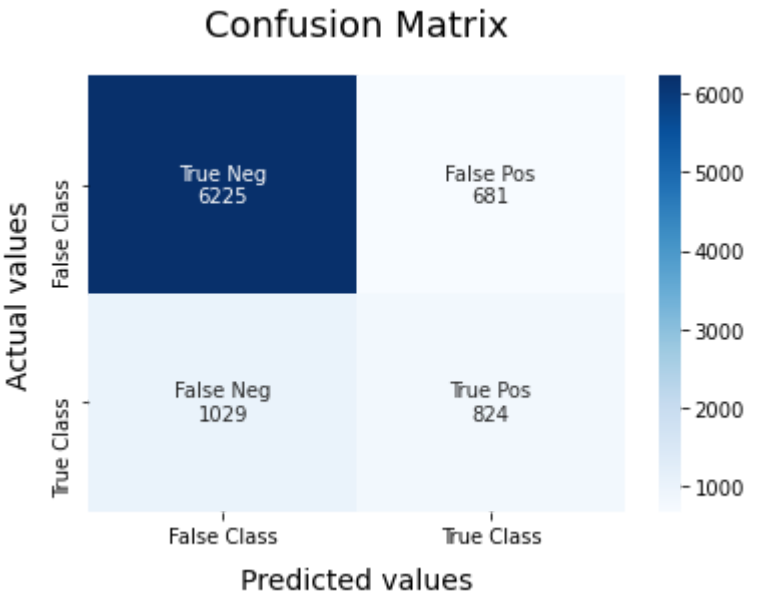
In [34]:
```
y_pred_bagging = clf.predict_proba(X_test)

# show bagging precision recall curve
best_bag_th, mean_bag_precision, std_bag_precision = plot_precision_recall_curve(y_test
```



In [35]:
```
print('Bagging Best Threshold=%f, G-Mean=%.3f, G-Std=%.3f' % (best_bag_th, mean_bag_pre
```

```
Bagging Best Threshold=0.444543, G-Mean=0.377, G-Std=0.067
```

In [36]:
```
# show bagging cm_matrix
plot_cm_matrix(y_test, y_pred_bagging.argmax(axis=1))
```



	precision	recall	f1-score	support
0	0.86	0.90	0.88	6906
1	0.55	0.44	0.49	1853
accuracy			0.80	8759
macro avg	0.70	0.67	0.69	8759
weighted avg	0.79	0.80	0.80	8759

Bonus: predict Kiba score

In [37]:
```
# split the data for train and valid
x_train, x_valid, y_train, y_valid = tts(X, train['KIBA'], test_size=0.2, random_state=
```

In [38]:
```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()

reg = Pipeline([
        ('ss', StandardScaler()),
        ('lr', lr),
    ])

reg.fit(x_train, y_train)
```

Out[38]:
```
Pipeline(steps=[('ss', StandardScaler()), ('lr', LinearRegression())])
```

In [39]:
```
# predict Kiba score
kipa_pred = reg.predict(x_valid)

kipa_pred
```

Out[39]:
```
array([11.43250236, 11.85683222, 11.75617044, ..., 11.99559288,
       11.804371  , 12.39216891])
```

Evaluate LR model

In [40]:
```
from sklearn.metrics import mean_squared_error

rmse = np.sqrt(mean_squared_error(y_valid, kipa_pred))

print('Linear Regression Root Mean Squared Error:', round(rmse, 3))
```

```
Linear Regression Root Mean Squared Error: 0.728
```

Next Step

- Testing

In [ ]:
```

```