Milestone3_performancemetrics

April 19, 2017

0.1 Description of our performance metrics

Multilabel classification

For our project, we have a multilabel classification problem involves mapping each movie in the dataset to a set of genre labels. In this type of classification problem, the labels are not mutually exclusive. For example, when classifying a movie into a set of genres, a single movie might be both Romance and Horror. Since the labels are not mutually exclusive, the predictions and true genre labels are now vectors of genre label sets, rather than vectors of genres. However, we can extend the fundamental ideas of precision, recall, etc. to operations on multilabel classification problems.

Multilabel evaluation metrics are grouped into two main categories: example based and label based metrics. Example based metrics are computed individually for each instance, then averaged to obtain the final value. Label based metrics are computed per label, instead of per instance. There are two approaches called micro-averaging and macro-averaging.

Let MLD be multilabel dataset, D is the number of samples, $\hat{y_j}$ is the predicted value for the j-th label sets of a given movie, y_j is the corresponsing true label sets, and n_{labels} is the number of labels.

We can calculate different measurements and evalute by multiple measurements:

1. Compare bit-wise . This can be too lenient, so we would not use it here.

2. Compare vector-ise (Accuracy classification score). This can be too strict, so we would not focus on it.

3. Jaccard similarity score: computes the average of Jaccard similarity coefficients (size of the intersection divided by the size of the union of two label sets) between pairs of label sets.

$$Jaccard(\hat{y_j}, y_j) = \frac{1}{|D|} \sum_{j=1}^{|D|} \frac{|\hat{y_j} \cap y_j|}{|\hat{y_j} \cup y_j|}$$

4. Hamming loss. It is the most common evaluation metric in the multilabel literature, computed as the symmetric difference between predicted and true labels and divided by the total number of labels in the MLD.

```
Then Hamming loss is defined as:
```

$$L_{Hamming}(\hat{y_j}, y_j) = \frac{1}{|D|} \sum_{j=1}^{|D|} \frac{xor(\hat{y_j}, y_j)}{n_{labels}}$$

The best value of Hamming loss is 0, the worst value of Hamming loss is 1.

5. mutil-label precision: This metric is computed as the ratio of relevant labels predicted by the classifier.

$$L_{precision}(\hat{y_j}, y_j) = \frac{1}{|D|} \sum_{j=1}^{|D|} \frac{|\hat{y_j} \cap y_j|}{|y_j|}$$

6. multi-label recall: It is a metric commonly used along with the previous one, measuring the proportion of predicted labels which are relevant.

$$L_{recall}(\hat{y_j}, y_j) = \frac{1}{|D|} \sum_{j=1}^{|D|} \frac{|\hat{y_j} \cap y_j|}{|\hat{y_j}|}$$

7. F1 score (Harmonic Mean of precision and recall): providing a balanced assessment between precision and sensitivity

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

For multi-label case, this is the specified weighted average of the F1 score of each class. In specific, we can specify the average in following ways: 'micro' calculates metrics globally by counting the total true positives, false negatives and false positives, 'macro' calculates metrics for each label, and find their unweighted mean, and 'weighted' calculates metrics for each label, and find their average, weighted by support (the number of true instances for each label) to account for imbalance.

If there are labels with more instances than others and if we want to bias our metric towards the most populated ones, we will use micro-average. If there are labels with more instances than others and if we want to bias your metric toward the least populated ones (or at least we don't want to bias toward the most populated ones), we will use macro-average. If the micro-average result is significantly lower than the macro-average one, it means that we have some gross misclassification in the most populated labels, whereas our smaller labels are probably correctly classified. If the macro-average result is significantly lower than the micro-average one, it means our smaller labels are poorly classified, whereas our larger ones are probably correctly classified. In terms of imbalance nature of data, we probably would like to use weighted-average to account for imbalance. The best value of F1 score is 1, the worst value of F1 score is 0.

Based on the literature review, **Hamming loss** (best:0, worst:1) and **F1 score** (best:1, worst:0) are the suggested metrics for multi-label classification problems. We will evaluate above performance evaluations during the model building process, but with a focus on Hamming loss and F1 score

0.2 Example of performance evaluation

Actual performance evaluation will be included in other ipython notebooks, the following code is just easier for other teammates.

```
In [2]: from sklearn.metrics import hamming_loss
        from sklearn.metrics import f1_score
        from sklearn.preprocessing import MultiLabelBinarizer
        import pandas as pd
        import numpy as np
        from sklearn import svm
        from sklearn.model_selection import cross_val_score
        from sklearn.model_selection import GridSearchCV
        from sklearn.metrics import make_scorer
```

```
In [85]: #For example purpose:
         imdb_example = pd.read_csv(r"C:\Users\cheriexu\Downloads\CS109B\movie_project\r\imdb_cluster_r
         imdb_example.head(5)
```

```
Out[85]:        X  certificates_R  certificates_PG  art.direction_1  \
         0    100               1                0         0.322495
         1  10001               0                1         0.027673
         2  10002               1                0         0.212394
         3  10003               0                1         0.019767
         4  10004               1                0         0.104764

            assistant.director_1  casting.director_1  cinematographer_1  \
         0              0.042103            0.010279           0.046254
```

```
1              0.093694           0.319431           0.165250
2              0.024906           0.006523           0.016308
3              0.024906           0.014133           0.025301
4              0.355406           0.017395           0.049812

   costume.department_1  costume.designer_1  countries_1  \
0              0.307966            0.028662     0.089642
1              0.307966            0.172860     0.014232
2              0.012453            0.005535     0.089642
3              0.126112            0.007116     0.536766
4              0.126112            0.040028     0.536766

                 ...             Action  Documentary  Musical  History  \
0                ...                  0            0        0        0
1                ...                  0            0        0        1
2                ...                  0            0        0        0
3                ...                  1            0        0        0
4                ...                  0            0        0        0

   Family  Fantasy  Sport  Biography  cluster_response  \
0       0        0      0          0                 4
1       0        0      0          0                 5
2       0        0      0          0                 1
3       0        0      0          0                 1
4       0        1      0          0                 1

                     genres_comb
0  "Romance", "Comedy", "Fantasy"
1  "Horror", "Thriller", "Action"
2   "Horror", "Thriller", "Drama"
3   "Horror", "Thriller", "Drama"
4   "Horror", "Thriller", "Drama"

[5 rows x 58 columns]
```

```
In [86]: #Input: dataframe from csv file
         #Output: y: response variable that is good for multi-label classification
         #        m: processor, may need to transform back in later
         def process_multilabel(dataframe):
             #convert response variable to a set format
             #for example, '"Romance, "Horror"' to ("Romance", "Horror")
             dataframe['genres_comb'] = dataframe['genres_comb'].apply(lambda x: eval(x))
             y = dataframe.ix[:,'genres_comb']
             m = MultiLabelBinarizer().fit(y)
             y = m.transform(y)
             return(y, m)
```

```
In [87]: y, m = process_multilabel(imdb_example)
```

```
In [91]: imdb_example.ix[:,'genres_comb'].head(5)
```

```
Out[91]: 0    (Romance, Comedy, Fantasy)
         1    (Horror, Thriller, Action)
         2     (Horror, Thriller, Drama)
         3     (Horror, Thriller, Drama)
         4     (Horror, Thriller, Drama)
         Name: genres_comb, dtype: object

In [92]: m.classes_

Out[92]: array(['Action', 'Adventure', 'Animation', 'Comedy', 'Drama', 'Family',
                'Fantasy', 'Horror', 'Music', 'Romance', 'Thriller'], dtype=object)

In [93]: #for example purpose
         y_true_m = y[1:100]
         y_pred_m = y[101:200]

In [94]: #two evluation functions:
         def f1score_evaluation(y_true_m, y_pred_m, average_method):
             #convert  = [Horror, Thriller, Action] to a list with binary indication
             #m = MultiLabelBinarizer().fit(y_true)
             #f1score = f1_score(m.transform(y_true),
             #     m.transform(y_pred),
             #     average= average_method)

             return(f1_score(y_true_m, y_pred_m, average = average_method))

         def hammingloss_evaluation(y_true_m, y_pred_m):
             #m = MultiLabelBinarizer().fit(y_true)
             #hammingloss = hamming_loss(m.transform(y_true),
             #     m.transform(y_pred))
              return(hamming_loss(y_true_m,
                 y_pred_m))

In [95]: f1score_evaluation(y_true_m, y_pred_m, 'weighted')

Out[95]: 0.49209799363385326

In [96]: hammingloss_evaluation(y_true_m, y_pred_m)

Out[96]: 0.2865013774104683
```

For cross_val_score or GridSearchCV:
f1 score with weighted average just set scoring ='f1_weighted' or:

```
eg:

clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
f1_scorer = make_scorer(f1score_evaluation, greater_is_better=Ture)
cross_val_score(clf, X, y, cv=3, scoring =f1_scorer)
```

hamming loss

```
hamming_scorer = make_scorer(hammingloss_evaluation, greater_is_better=False)
cross_val_score(clf, X, y, cv=3, scoring =hamming_scorer)
```

Milestone3_preidct_multilabel_final

April 19, 2017

In this python notebook, we use skmultilearn package as it focuses on multilabel classification. We input the processeed data and use the training set to train the model and examine performance metrics for the test set.

0.0.1 Multi-label classification strategy

We will use multi-label classification. Before we discuss which classifier to use (ex. KNN, SVM), we should consdier how we will treat the response variable first.

The "strategies" we used is listed below: 1. Binary Relevence(BR) we seperate each genre into seperate problems (one for each genre). However, this ignore label dependence. Ex. if a movie is tagged as Drama, it is likely that it is also tagged as Action. if a movie is tagged as Horror, it is likely that it is also tagged as Romance. If two classes of a genre (Yes/No) have very uneven sizes in the training set, the classifier will lean toward the class with higher movie number. There is a method called a label correction strategy that can help to improve accuracy For example, if our prediction is [Y_horror, Y_romance, Y_drama]= [1,1,0], which does not really happen in training set. We find another likely matching vector. We may change our prediction to be [1,0,1].

2. Classifier Chains (CC) We seperate each genre into seperate problems, but include previous predictions as predictors. For example, X is our predictor for Y_horror. Next, X, Y_horror are our predictor for Y_romance. However, error may be propagated down the chain.

3. Label Powerset (LP) Instead of having seperate Y_i for each genre i, we will predict only Y. Y has 2ˆI possible values where I is the number of genre. For example, if Y_horror = 1, Y_romance = 0, Y_drama = 1, Y = [101] However, imbalance of the data can be an issue.

0.0.2 Classifier

For each of the strategy, we will then apply different classifier. 1. KNN 2. SVM

0.0.3 Performance mertic

Please refer to the python notebook "Milestone3_performancemetrics". In short, we will evaluate majorly based on F1 score and Hamming loss.

```
In [1]: import os
        import numpy as np
        import pandas as pd
        import matplotlib
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        import matplotlib.cm as cmx
        import matplotlib.colors as colors
        import math
        import seaborn.apionly as sns
        import datetime as dt
        from sklearn.metrics import hamming_loss
        from sklearn.metrics import accuracy_score
        from sklearn.metrics import f1_score
        from sklearn.metrics import precision_score
        from sklearn.metrics import jaccard_similarity_score

        from skmultilearn.problem_transform import BinaryRelevance, LabelPowerset, ClassifierChain
        from sklearn.naive_bayes import GaussianNB
        from skmultilearn.ensemble.rakeld import RakelD

        from sklearn.model_selection import GridSearchCV
        from sklearn.naive_bayes import MultinomialNB

        # classifier
        from skmultilearn.adapt import MLkNN
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.ensemble import ExtraTreesClassifier
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.svm import SVC, LinearSVC

        from functools import partial

        # cross validation
        from sklearn.model_selection import KFold
        from sklearn.model_selection import StratifiedKFold

        %matplotlib inline
```

```
In [2]: dir_python_notebook = os.getcwd()
        dir_movie_project = os.path.abspath(os.path.join(dir_python_notebook, os.pardir))
        dir_data = os.path.join(dir_movie_project, 'project')
```

1 Global variable

```
In [3]: MODELS = {
            "Gaussian Naive Bayes": GaussianNB(),
            #"Random Forest": RandomForestClassifier(random_state=0),
            #"Extra Trees": ExtraTreesClassifier(n_estimators=100, random_state=0),
            "SVM": SVC(),
            "KNN, k=5": KNeighborsClassifier(n_neighbors=5),
            #"KNN, k=10": KNeighborsClassifier(n_neighbors=10),
        }
```

```
In [4]: STRATEGIES = {
             'Binary Relevance': BinaryRelevance(),
            'Classifier Chain': ClassifierChain(),
            'Label Powerset': LabelPowerset()

        }
```

2 Data

2.1 sample

We originall have downloaded the movies with TMDB id from 1 to 300000. 1. many id in between that range actually are not valid (There is no movie for that tmdb id) 2. We only include movies that have data in both TMDB and IMDB. 3. We exclude movies that have more than 50% of features missing. Overall, we have 68186 movies.

2.2 predictior variable

1. We originally downloaded all 51 features from IMDB
2. We removes features that have missing rate > 50% for the samples
3. Some features has list of values, such as cast. As the list is ranked by columns, we convert "cast" to "cast1", "cast2", etc, so each column will only hold 1 value , instead of a list of values.
4. We impute the missing data.
5. We conver categorical variable such as "certificates_R" to relative frequency, so we can apply a lot of methods that may only be suitable for numeric values.
6. We select important features by methods like LASSO and Random Forest. Overall, we have 31 predictor variables.

2.3 response variable

1. We use IMDB genre and convert it to 28 columns. They are binary. Ex. if movies has genre of "Horror" and "Crime", the relevent columns will be 1, else 0.
2. Because 28 genres are too many, we decide to reduce the response variable by clustering with methods like K-mean, etc. We have 7 clusters. We also examine the cluster composition to ensure they make senese. Ex. 1 cluster include only "Horror" and "Thriller". 1 cluster includes "Crime", "Mystery","Film.Noir". Overall, we have 7 response variables.

```
In [5]: filename = dir_data + '//imdb_cluster_result_whole.csv'
        data_df= pd.read_csv(filename)

In [6]: data_df.shape

Out[6]: (68186, 66)

In [7]: data_df.columns

Out[7]: Index([u'certificates_R', u'certificates_PG', u'art.direction_1',
               u'assistant.director_1', u'cinematographer_1', u'costume.department_1',
               u'costume.designer_1', u'countries_1', u'director_1', u'distributors_1',
               u'editor_1', u'languages_1', u'make.up_1', u'miscellaneous.companies_1',
               u'miscellaneous.crew_1', u'original.music_1', u'producer_1',
               u'production.companies_1', u'production.manager_1', u'sound.crew_1',
               u'writer_1', u'special.effects.companies_1', u'cast_1', u'cast_2',
               u'cast_3', u'cast_4', u'runtimes_avg', u'rating', u'imdb_id',
               u'tmdb_id', u'Sci.Fi', u'Crime', u'Romance', u'Animation', u'Music',
               u'Adult', u'Comedy', u'War', u'Horror', u'Film.Noir', u'Western',
               u'News', u'Reality.TV', u'Thriller', u'Adventure', u'Mystery', u'Short',
               u'Talk.Show', u'Drama', u'Action', u'Documentary', u'Musical',
               u'History', u'Family', u'Fantasy', u'Game.Show', u'Sport', u'Biography',
               u'cluster_response', u'cluster_1', u'cluster_2', u'cluster_3',
               u'cluster_4', u'cluster_5', u'cluster_6', u'cluster_7'],
              dtype='object')
```

```
In [8]: X_var= list(data_df.columns.values)
        X_var = X_var[0:28]
        print(len(X_var))
        print(X_var)
```

```
28
['certificates_R', 'certificates_PG', 'art.direction_1', 'assistant.director_1', 'cinematographer_1', 'co
```

```
In [9]: Y_var = list(data_df.columns.values)
        Y_var = Y_var[59:66]
        print(len(Y_var))
        print(Y_var)
```

```
7
['cluster_1', 'cluster_2', 'cluster_3', 'cluster_4', 'cluster_5', 'cluster_6', 'cluster_7']
```

3 Prediction

```
In [10]: def get_metric_data_frame(Y_true_train, y_pred_train,train, model_name, strategy):
             metric_train = {}
             metric_train["micro-f1"] = f1_score(Y_true_train, y_pred_train, average="micro")
             metric_train["weighted-f1"] = f1_score(Y_true_train, y_pred_train, average="weighted")
             metric_train["samples-f1"] = f1_score(Y_true_train, y_pred_train, average="samples")
             metric_train["macro-f1"] = f1_score(Y_true_train, y_pred_train, average="macro")
             metric_train["hamming_loss"] = hamming_loss(Y_true_train, y_pred_train)
             metric_train["subset_accuracy"] = accuracy_score(Y_true_train, y_pred_train)
             metric_train["jaccard"] = jaccard_similarity_score(Y_true_train, y_pred_train)

             metric_test_df_new = pd.DataFrame.from_dict(metric_train, orient='index').transpose()
             metric_test_df_new['model'] = model_name
             metric_test_df_new['strategy'] = strategy
             metric_test_df_new['train_test'] = train

             return metric_test_df_new
```

```
In [11]: def get_predicion_result(X_train, Y_true_train, X_test, Y_true_test, strategy, model, model_na

             clf = BinaryRelevance(model)
             if (strategy == "Classifier Chain"):
                 clf = ClassifierChain(model)
             if (strategy == "Label Powerset"):
                 clf = LabelPowerset(model)
             # train
             clf.fit(X_train, Y_true_train)

             # predict
             y_pred_train = clf.predict(X_train)
             y_pred_test = clf.predict(X_test)


             metric_df = get_metric_data_frame(Y_true_train, y_pred_train, "train", model_name, strateg
             metric_df  = metric_df.append(get_metric_data_frame(Y_true_test, y_pred_test, "test", mode

             return metric_df
```

```
In [12]: def predict(X_train, Y_true_train, X_test, Y_true_test, model_name_list, strategy_list):
             count = 0
             for i in range(len(model_name_list)):
                 model_name = model_name_list[i]
                 model = MODELS[model_name]
                 for strategy in strategy_list:

                     metric_df_new = get_predicion_result(X_train, Y_true_train, X_test, Y_true_test,st
                     if count > 0 :
                         metric_df = metric_df.append(metric_df_new, ignore_index=True)
                     else:
                         metric_df = metric_df_new

                     count = count + 1

             return metric_df
```

3.1 Tuning

We examinf KNN and SVM. To use the prediction methods, plesae add the new model and the optimal parameter in the dictionary MODELS.

3.1.1 Resource

Cross validation example from scikit-multilearn: http://scikit.ml/api/loading.html#cross-validation-and-train-test-splits

Tuning parameter example from scikit-multilearn: http://scikit.ml/api/model_estimation.html#estimating-hyper-parameter-k-for-embedded-classifiers

```
In [13]: train_df = data_df[data_df[u'tmdb_id'] < 100000]
         test_df = data_df[data_df[u'tmdb_id'] >= 100000]
         X_train = train_df[X_var]
         Y_true_train = train_df[Y_var]
         X_test = test_df[X_var]
         Y_true_test = test_df[Y_var]
```

3.2 Tuning for KNN

```
In [14]: # find out what parameters can we tune
         KNeighborsClassifier().get_params().keys()

         # I choose to only tune n_neightbors

Out[14]: ['n_neighbors',
          'n_jobs',
          'algorithm',
          'metric',
          'metric_params',
          'p',
          'weights',
          'leaf_size']

In [15]: parameters = {
             'labelset_size': [7],
             'classifier': [BinaryRelevance()],
             'classifier__classifier': [KNeighborsClassifier()],
             'classifier__classifier__n_neighbors': [1, 9],
         }

         clf = GridSearchCV(RakelD(), parameters, scoring='f1_macro')
         clf.fit(X_train, Y_true_train)

         print clf.best_params_, clf.best_score_
```

```
{'labelset_size': 7, 'classifier__classifier': KNeighborsClassifier(algorithm='auto', leaf_size=30, metri
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'), 'classifier': BinaryRelevance(classifier=KNeighborsClassifier(algorithm=
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'),
        require_dense=[True, True]), 'classifier__classifier__n_neighbors': 1} 0.365419276444
```

```
In [16]: parameters = {
             'labelset_size': [7],
             'classifier': [ClassifierChain()],
             'classifier__classifier': [KNeighborsClassifier()],
             'classifier__classifier__n_neighbors': [1, 9],
         }

         clf = GridSearchCV(RakelD(), parameters, scoring='f1_macro')
         clf.fit(X_train, Y_true_train)

         print clf.best_params_, clf.best_score_
```

```
{'labelset_size': 7, 'classifier__classifier': KNeighborsClassifier(algorithm='auto', leaf_size=30, metri
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'), 'classifier': ClassifierChain(classifier=KNeighborsClassifier(algorithm=
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'),
        require_dense=[True, True]), 'classifier__classifier__n_neighbors': 1} 0.365419276444
```

```
In [17]: parameters = {
             'labelset_size': [7],
             'classifier': [LabelPowerset()],#[LabelPowerset(), BinaryRelevance()],
             'classifier__classifier': [KNeighborsClassifier()],
             'classifier__classifier__n_neighbors': [1, 9],
         }

         clf = GridSearchCV(RakelD(), parameters, scoring='f1_macro')
         clf.fit(X_train, Y_true_train)

         print clf.best_params_, clf.best_score_
```

```
{'labelset_size': 7, 'classifier__classifier': KNeighborsClassifier(algorithm='auto', leaf_size=30, metri
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'), 'classifier': LabelPowerset(classifier=KNeighborsClassifier(algorithm='a
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform'),
       require_dense=[True, True]), 'classifier__classifier__n_neighbors': 1} 0.365419276444
```

3.3 Tuning for SVM

A smaller dataset was used here for tuning

```
In [18]: SVC().get_params().keys()

Out[18]: ['kernel',
          'C',
          'verbose',
          'probability',
          'degree',
          'shrinking',
          'max_iter',
          'decision_function_shape',
          'random_state',
          'tol',
          'cache_size',
          'coef0',
          'gamma',
          'class_weight']

In [19]: df_new = data_df[data_df[u'tmdb_id'] < 10000]
         X_train1 = df_new[X_var]
         Y_true_train1 = df_new[Y_var]

In [20]: parameters = {
             'labelset_size': [7],
             'classifier': [BinaryRelevance()],
             'classifier__classifier': [SVC()],
             'classifier__classifier__C': [1, 10, 100],
             'classifier__classifier__gamma': [0.1, 0.01, 0.001]
         }

         clf = GridSearchCV(RakelD(), parameters, scoring='f1_macro')
         clf.fit(X_train1, Y_true_train1)

         print clf.best_params_, clf.best_score_

/Users/aixu/anaconda/lib/python2.7/site-packages/sklearn/metrics/classification.py:1113: UndefinedMetri
  'precision', 'predicted', average, warn_for)

{'classifier__classifier__gamma': 0.1, 'labelset_size': 7, 'classifier__classifier': SVC(C=100, cache_size
  decision_function_shape=None, degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False), 'classifier': BinaryRelevance(classifier=SVC(C=100, cache_size=200, class_w
  decision_function_shape=None, degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False),
        require_dense=[True, True]), 'classifier__classifier__C': 100} 0.369298212185

In [21]: parameters = {
             'labelset_size': [7],
             'classifier': [ClassifierChain()],
             'classifier__classifier': [SVC()],
             'classifier__classifier__C': [1, 10, 100],
             'classifier__classifier__gamma': [0.1, 0.01, 0.001]
         }

         clf = GridSearchCV(RakelD(), parameters, scoring='f1_macro')
         clf.fit(X_train1, Y_true_train1)

         print clf.best_params_, clf.best_score_
```

```
{'classifier__classifier__gamma': 0.1, 'labelset_size': 7, 'classifier__classifier': SVC(C=100, cache_size
  decision_function_shape=None, degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False), 'classifier': ClassifierChain(classifier=SVC(C=100, cache_size=200, class_w
  decision_function_shape=None, degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False),
        require_dense=[True, True]), 'classifier__classifier__C': 100} 0.391546701311
```

```
In [22]: parameters = {
             'labelset_size': [7],
             'classifier': [LabelPowerset()],
             'classifier__classifier': [SVC()],
             'classifier__classifier__C': [1, 10, 100],
             'classifier__classifier__gamma': [0.1, 0.01, 0.001]
         }

         clf = GridSearchCV(RakelD(), parameters, scoring='f1_macro')
         clf.fit(X_train1, Y_true_train1)

         print clf.best_params_, clf.best_score_
```

```
{'classifier__classifier__gamma': 0.1, 'labelset_size': 7, 'classifier__classifier': SVC(C=100, cache_size
  decision_function_shape=None, degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False), 'classifier': LabelPowerset(classifier=SVC(C=100, cache_size=200, class_wei
  decision_function_shape=None, degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False),
       require_dense=[True, True]), 'classifier__classifier__C': 100} 0.39160233552
```

```
In [14]: # After tuning, we input the variable below

         MODELS = {
             #"Gaussian Naive Bayes": GaussianNB(),
             #"Random Forest": RandomForestClassifier(random_state=0),
             #"Extra Trees": ExtraTreesClassifier(n_estimators=100, random_state=0),
             "SVM C=100 gamma=0.1": SVC(C=100, gamma=0.1),
             "KNN k=1": KNeighborsClassifier(n_neighbors=1)
         }
```

4 Cross-validation

We tried both traditional k fold and stratified k fold.

```
In [15]: def cross_validate_by_kfold(data_df, X_var, Y_var, n_split):
             count = 0
             # remember to set n_splits and shuffle!
             kf = KFold(n_splits=n_split, random_state=None, shuffle= True)

             X = data_df[X_var]
```

```
        Y = data_df[Y_var]

        for train_index, test_index in kf.split(X, Y):
            X_train = X.iloc[train_index]
            Y_true_train = Y.iloc[train_index]

            X_test = X.iloc[test_index]
            Y_true_test = Y.iloc[test_index]

            model_name_list = MODELS.keys()

            strategy_list = STRATEGIES.keys()

            metric_df_new = predict(X_train, Y_true_train, X_test, Y_true_test, model_name_list, s

            if count == 0 :
                metric_df = metric_df_new
            else:
                metric_df = metric_df.append(metric_df_new, ignore_index=True)
            count = count + 1

        metric_grouped_df = metric_df.groupby(['model','strategy','train_test']).mean().reset_inde

        return metric_grouped_df
```

```
In [16]: def cross_validate_by_stratifiedkfold(data_df, X_var, Y_var, n_split):
        count = 0

        strategy_list = STRATEGIES.keys()

        for strategy in strategy_list:

            sc = STRATEGIES[strategy]

            # remember to set n_splits and shuffle!
            kf = StratifiedKFold(n_splits=n_split, random_state=None, shuffle=True)

            X = data_df[X_var]
            Y = data_df[Y_var]


            for train_index, test_index in kf.split(X, sc.transform(Y)):

                X_train = X.iloc[train_index]
                Y_true_train = Y.iloc[train_index]

                X_test = X.iloc[test_index]
                Y_true_test = Y.iloc[test_index]

                model_name_list = MODELS.keys()

                #strategy_list = STRATEGIES.keys()
                strategy_list = [strategy]
                metric_df_new = predict(X_train, Y_true_train, X_test, Y_true_test, model_name_lis

                if count == 0 :
                    metric_df = metric_df_new
                else:
                    metric_df = metric_df.append(metric_df_new, ignore_index=True)

                count = count + 1

        metric_grouped_df = metric_df.groupby(['model','strategy','train_test']).mean().reset_inde

        return metric_grouped_df
```

```
In [17]: # test with a smaller data set to ensure the method works
         data_df_new = data_df[data_df[u'tmdb_id'] < 10000]
         count = 0
         # remember to set n_splits and shuffle!
         n_split=5
         kf = KFold(n_splits=n_split, random_state=None, shuffle= True)

         X = data_df_new[X_var]
         Y = data_df_new[Y_var]

         for train_index, test_index in kf.split(X, Y):
             X_train = X.iloc[train_index]
             Y_true_train = Y.iloc[train_index]

             X_test = X.iloc[test_index]
             Y_true_test = Y.iloc[test_index]

             model_name_list = MODELS.keys()

             strategy_list = STRATEGIES.keys()

             metric_df_new = predict(X_train, Y_true_train, X_test, Y_true_test, model_name_list, strat

             if count == 0 :
                 metric_df = metric_df_new
             else:
                 metric_df = metric_df.append(metric_df_new, ignore_index=True)
             count = count + 1

         metric_grouped_df = metric_df.groupby(['model','strategy','train_test']).mean().reset_index()
         metric_grouped_df
```

```
/Users/aixu/anaconda/lib/python2.7/site-packages/sklearn/metrics/classification.py:1113: UndefinedMetri
  'precision', 'predicted', average, warn_for)
```

```
Out[17]:                  model          strategy train_test  micro-f1   jaccard  \
         0              KNN k=1  Binary Relevance       test  0.505602  0.387116
         1              KNN k=1  Binary Relevance      train  1.000000  1.000000
         2              KNN k=1  Classifier Chain       test  0.505602  0.387116
         3              KNN k=1  Classifier Chain      train  1.000000  1.000000
         4              KNN k=1    Label Powerset       test  0.505602  0.387116
         5              KNN k=1    Label Powerset      train  1.000000  1.000000
```

```
6   SVM C=100 gamma=0.1  Binary Relevance       test  0.537955  0.430592
7   SVM C=100 gamma=0.1  Binary Relevance      train  0.763547  0.678921
8   SVM C=100 gamma=0.1  Classifier Chain       test  0.528095  0.422838
9   SVM C=100 gamma=0.1  Classifier Chain      train  0.770704  0.711194
10  SVM C=100 gamma=0.1    Label Powerset       test  0.521780  0.412756
11  SVM C=100 gamma=0.1    Label Powerset      train  0.871918  0.838688

    macro-f1  samples-f1  subset_accuracy  weighted-f1  hamming_loss
0   0.398596    0.499406         0.094591     0.505252      0.322716
1   1.000000    1.000000         1.000000     1.000000      0.000000
2   0.398596    0.499406         0.094591     0.505252      0.322716
3   1.000000    1.000000         1.000000     1.000000      0.000000
4   0.398596    0.499406         0.094591     0.505252      0.322716
5   1.000000    1.000000         1.000000     1.000000      0.000000
6   0.373204    0.543016         0.134830     0.500775      0.272829
7   0.689827    0.756285         0.439813     0.744346      0.139355
8   0.402096    0.530921         0.137373     0.514970      0.294041
9   0.722451    0.769975         0.544403     0.764564      0.143154
10  0.398465    0.522800         0.125347     0.511512      0.302629
11  0.852062    0.869951         0.754278     0.870238      0.081563
```

5 Get performance metrics

We decided to use **weighted-f1** and **hamming_loss** as our final performance metrics

```
In [70]: def get_metric_data_frame(Y_true_train, y_pred_train,train, model_name, strategy):
             metric_train = {}

             metric_train["weighted-f1"] = f1_score(Y_true_train, y_pred_train, average="weighted")
             metric_train["hamming_loss"] = hamming_loss(Y_true_train, y_pred_train)

             metric_test_df_new = pd.DataFrame.from_dict(metric_train, orient='index').transpose()
             metric_test_df_new['model'] = model_name
             metric_test_df_new['strategy'] = strategy
             metric_test_df_new['train_test'] = train

             return metric_test_df_new
```

```
In [103]: data_df_2 = data_df[data_df[u'tmdb_id'] < 50000]
```

We used a smaller dataset (half of original size) here since the runtime for whole dataset is too long. We'll re-run the whole dataset on AWS in following milestones

```
In [104]: # test
          n_split = 5
          metric_df_kfold  = cross_validate_by_kfold(data_df_2, X_var, Y_var, n_split)
```

```
In [105]: metric_df_stratifiedkfold  = cross_validate_by_kfold(data_df_2, X_var, Y_var, n_split)
```

```
In [106]: metric_df_kfold.head(3)
```

```
Out[106]:      model          strategy train_test  micro-f1   jaccard  macro-f1  \
          0  KNN k=1  Binary Relevance       test  0.506396  0.387215  0.399619
          1  KNN k=1  Binary Relevance      train  1.000000  1.000000  1.000000
          2  KNN k=1  Classifier Chain       test  0.506396  0.387215  0.399619
```

```
   samples-f1  subset_accuracy  weighted-f1  hamming_loss
0    0.499167         0.096671     0.506243      0.322551
1    1.000000         1.000000     1.000000      0.000000
2    0.499167         0.096671     0.506243      0.322551
```

In [107]:
```
metric_df_stratifiedkfold.head(3)
```

Out[107]:
```
      model           strategy train_test  micro-f1   jaccard  macro-f1  \
0  KNN k=1  Binary Relevance       test  0.508331  0.388699  0.401999
1  KNN k=1  Binary Relevance      train  1.000000  1.000000  1.000000
2  KNN k=1  Classifier Chain       test  0.508331  0.388699  0.401999

   samples-f1  subset_accuracy  weighted-f1  hamming_loss
0    0.500738          0.09459     0.508198      0.321129
1    1.000000          1.00000     1.000000      0.000000
2    0.500738          0.09459     0.508198      0.321129
```

6 Visualization on performance

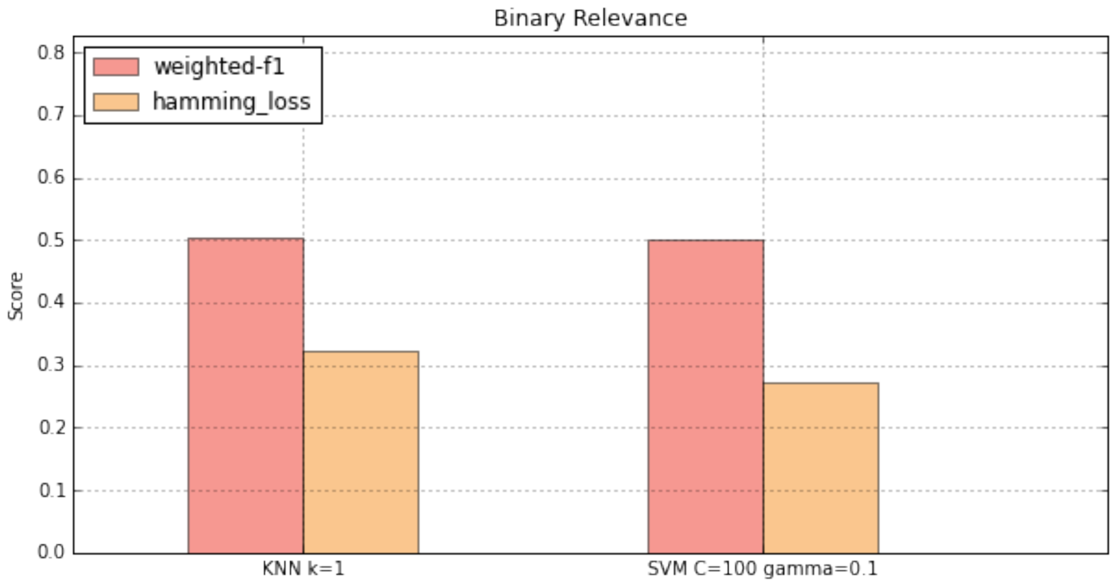
In [18]:
```
def plot_metric_for_strategy(train_test, strategy, metric):
    title = metric + " for " + strategy
    metric_plot_df = metric_df[(metric_df['strategy']== strategy) & (metric_df['train_test']==
    fig, ax = plt.subplots(1, 1, figsize=(8,4))
    ax = sns.barplot(x="model", y=metric, data=metric_plot_df)
    xt = plt.xticks(rotation=45)
    ax.set_title(title)
    plt.show()
```

In [19]:
```
metric_df = metric_df_stratifiedkfold
```

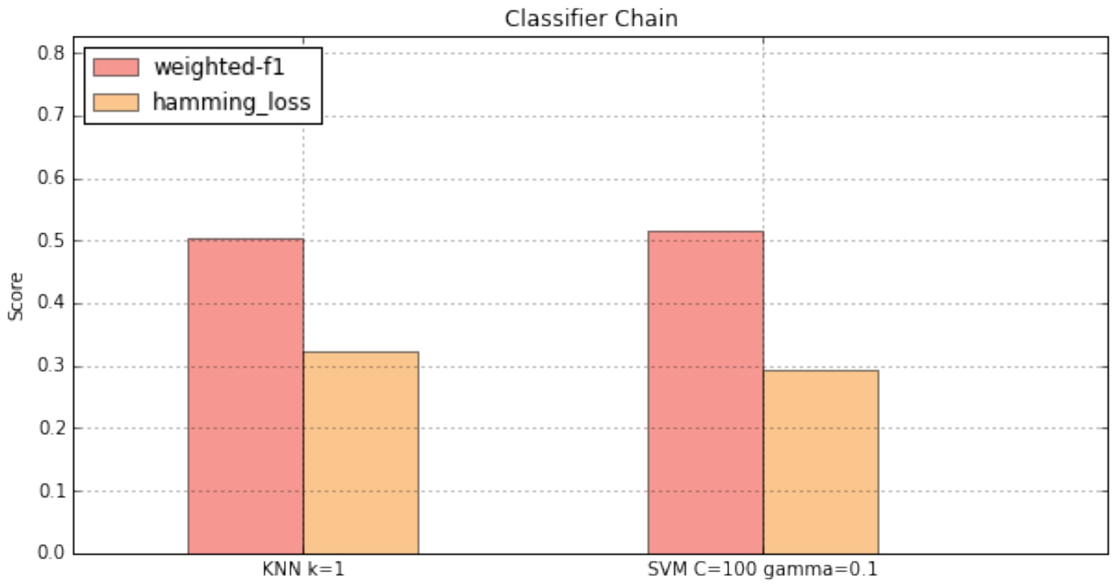
1. Compare Metrics (Weighted-F1, Hamming_Loss)

In [109]:
```
def plot_comparison(strategy, metric1, metric2):
    metric_plot_df = metric_df[(metric_df['strategy']== strategy) & (metric_df['train_test']=
    # Setting the positions and width for the bars
    pos = list(range(len(metric_plot_df['model'])))
    width = 0.25
    # Plotting the bars
    fig, ax = plt.subplots(figsize=(10,5))
    # Create a bar with pre_score data,
    # in position pos,
    plt.bar(pos,

        metric_plot_df[metric1],
        # of width
        width,
        # with alpha 0.5
        alpha=0.5,
        # with color
        color='#EE3224',
        # with label the first value in first_name
        label='KNN k=1')
    # Create a bar with mid_score data,
    # in position pos + some width buffer,
    plt.bar([p + width for p in pos],

        metric_plot_df[metric2],
        # of width
        width,
        # with alpha 0.5
        alpha=0.5,
        # with color
        color='#F78F1E',
        # with label the second value in model
        label='SVM C=100 gamma=0.1')

    # Set the y axis label
    ax.set_ylabel('Score')

    # Set the chart's title
    ax.set_title(strategy)

    # Set the position of the x ticks
    ax.set_xticks([p +  width for p in pos])

    # Set the labels for the x ticks
    ax.set_xticklabels(metric_plot_df['model'].values)

    # Setting the x-axis and y-axis limits
    plt.xlim(min(pos)-width, max(pos)+width*4)
    plt.ylim([0, max(metric_plot_df[metric1] + metric_plot_df[metric2])])

    # Adding the legend and showing the plot
    plt.legend([metric1, metric2], loc='upper left')
    plt.grid()
    plt.show()
```

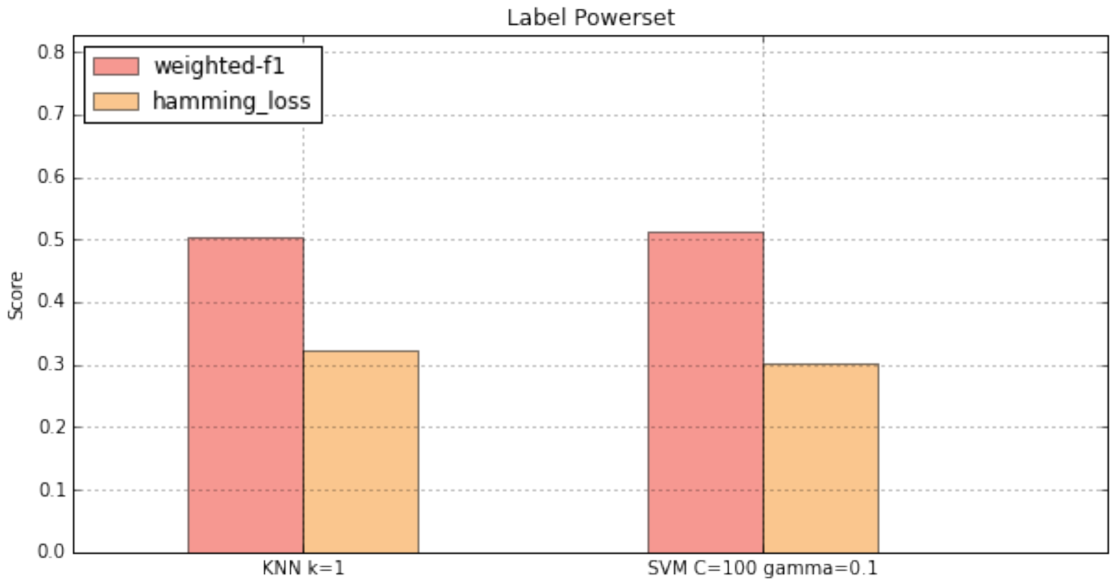
```
In [110]: plot_comparison('Binary Relevance', 'weighted-f1','hamming_loss')
```



```
In [111]: plot_comparison('Classifier Chain','weighted-f1','hamming_loss')
```

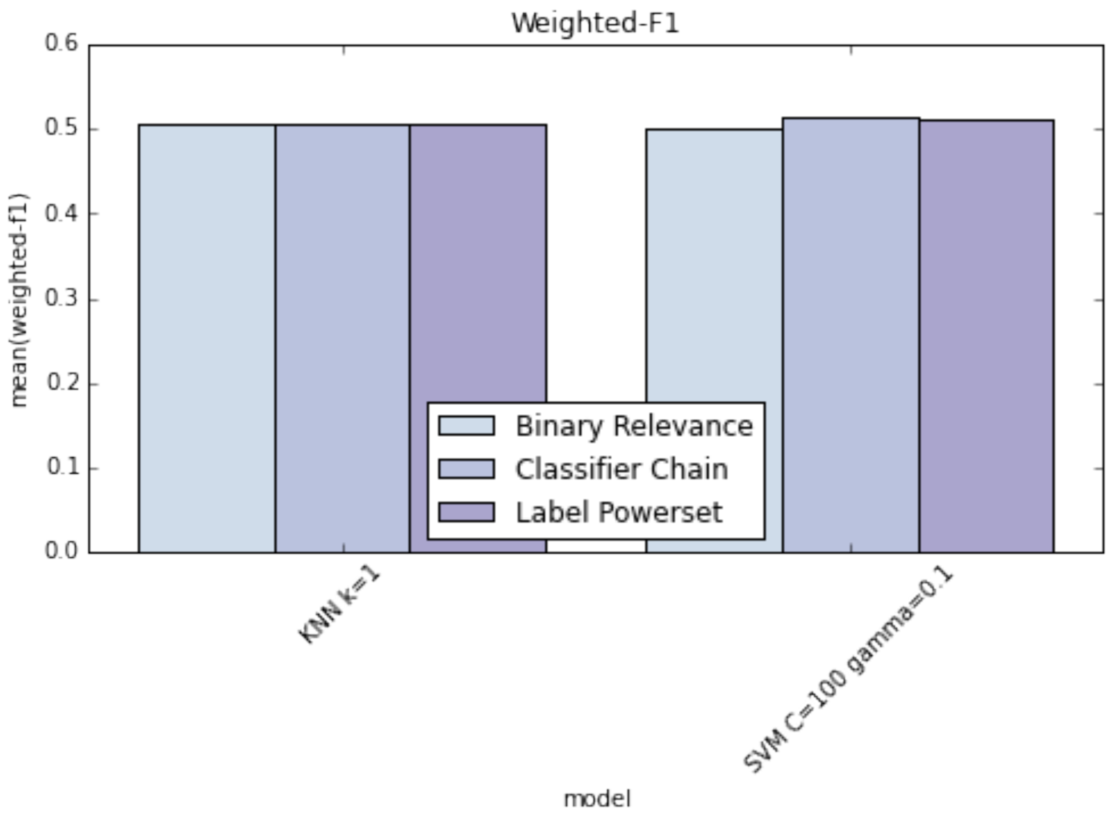


```
In [112]: plot_comparison('Label Powerset','weighted-f1','hamming_loss')
```

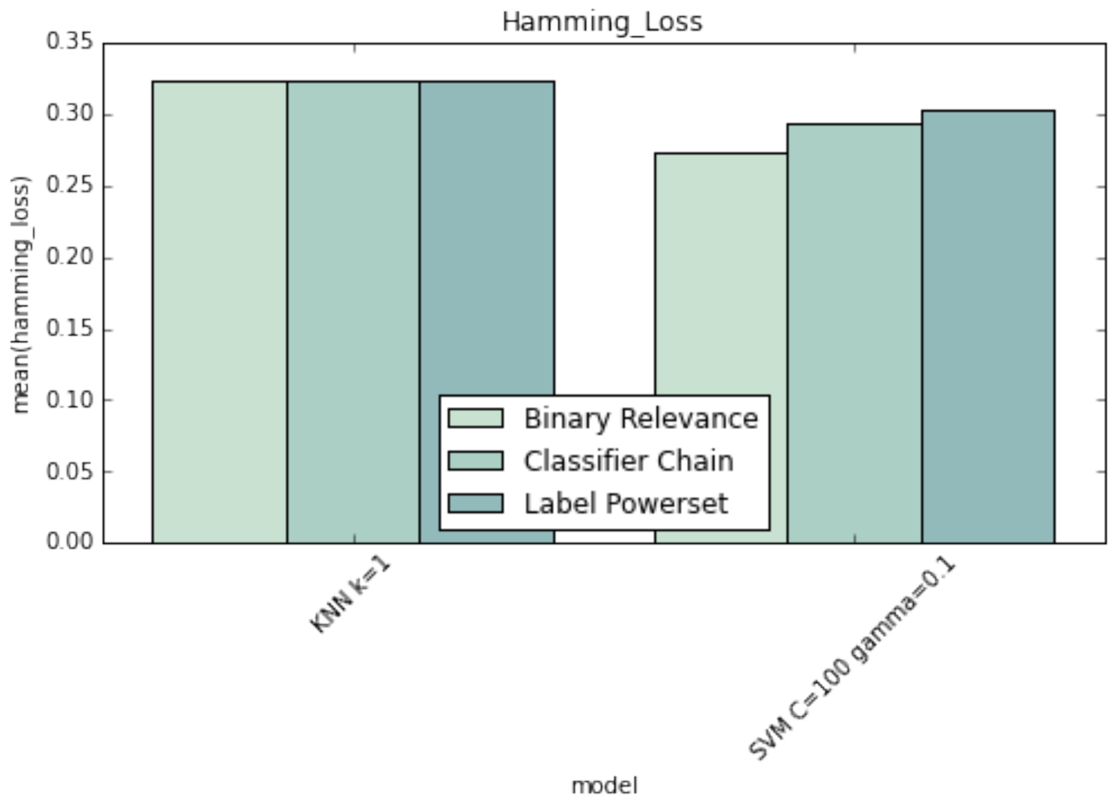


2. Compare Strategies (Binary Relevance, Classifier Chain, Label Powerset)

```
In [101]: metric_plot_df = metric_df[(metric_df['train_test']== 'test')]
          fig, ax = plt.subplots(1, 1, figsize=(8,4))
          sns.set_palette(sns.cubehelix_palette(8,start=1, rot=-.5))
          ax = sns.barplot(x="model", y="weighted-f1", hue="strategy", data=metric_plot_df)
          xt = plt.xticks(rotation=45)
          ax.set_title("Weighted-F1")
          plt.legend(loc='best')
          plt.show()
```

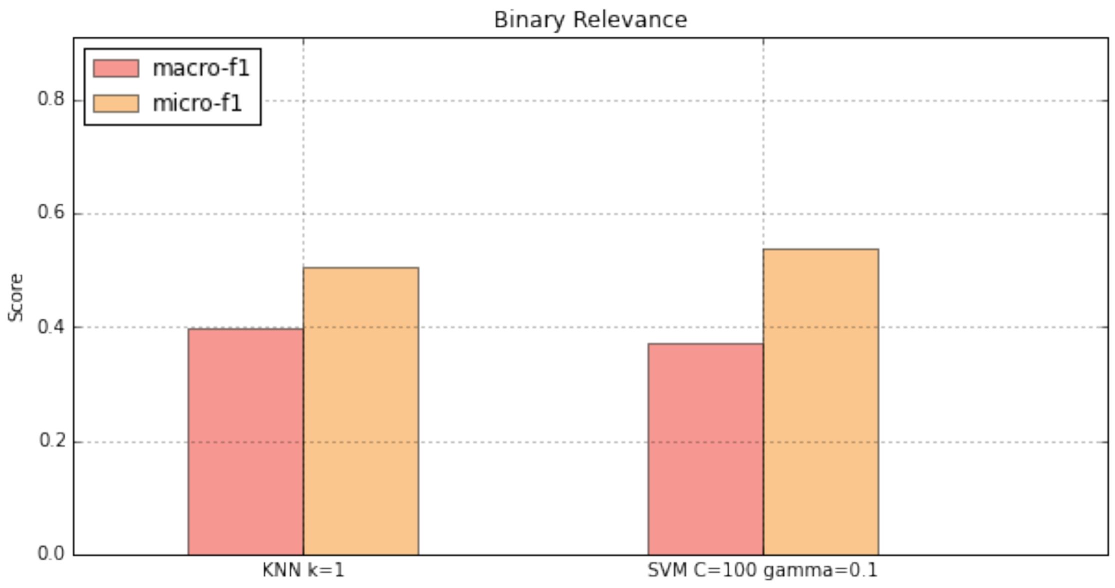


```
In [100]: metric_plot_df = metric_df[(metric_df['train_test']== 'test')]
          fig, ax = plt.subplots(1, 1, figsize=(8,4))
          sns.set_palette(sns.cubehelix_palette(10,start=1, rot=-.75))
          ax = sns.barplot(x="model", y="hamming_loss", hue="strategy", data=metric_plot_df)
          xt = plt.xticks(rotation=45)
          ax.set_title("Hamming_Loss")
          plt.legend(loc='best')
          plt.show()
```

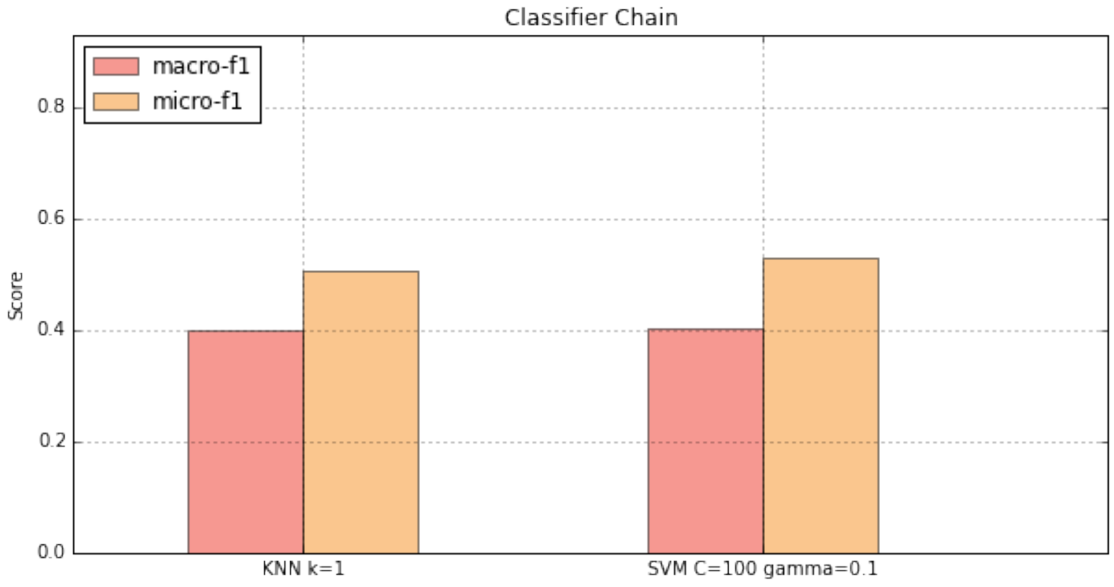


3. Compare Macro f1 vs Micro f1

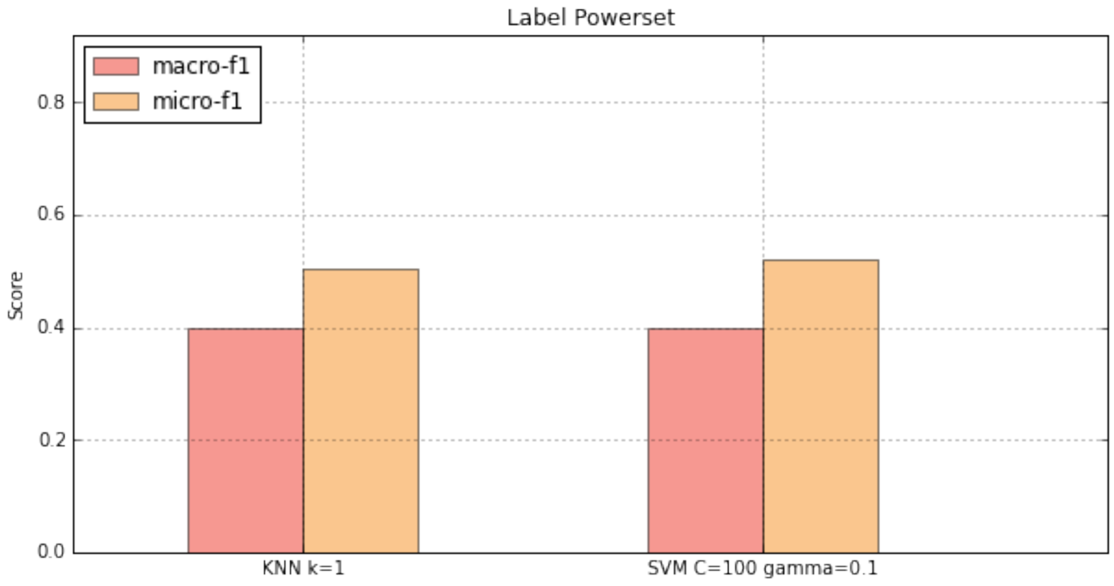
```
In [113]: plot_comparison('Binary Relevance', 'macro-f1','micro-f1')
```



```
In [114]: plot_comparison('Classifier Chain', 'macro-f1','micro-f1')
```



```
In [115]: plot_comparison('Label Powerset', 'macro-f1','micro-f1')
```



7 Results and Discussion

7.1 Goal

1. We examine the performance metric for the test set among various classifiers.

- For hamming loss, the smaller the value , the smaller the difference between predicted and true labels.
- For F1 score, the larger the value , the smaller the difference between predicted and true labels. We will focus on weighted F1 score for now.

2. We examine the performance metric for each strategy :Binary Relevance,Classifier Chain,Label Powerset.
3. We will look into the performance metric for labels with more movies vs labels with fewer moviews.
   - We can examine the difference between F1 score for macro average and micro average. If the micro-average result is significantly lower than the macro-average one, it means that we have some gross misclassification in the most populated labels, whereas our smaller labels are probably correctly classified.
4. We examine if cross-validation by kfold has similiar result as by stratified kfold
   - k-folding may lead to severe problems with label combination representability across folds, thus if the data exhibits a strong label co-occurrence structure, using a label-combination based stratified k-fold will be better.

7.2 1. Classifier with the best metric

After comparing KNN and SVM based on weighted-f1 and hamming_loss, we found out the **SVM** has lower hamming_loss scores for all 3 strategies, and weighted-f1 for both classifier are similiar. The performances of KNN and SVM does not differ much. Generally, **SVM** is better. As we only tuned limited set of parameters for SVM here, we may need to re-do the parameter tuning on whole dataset to get more accurate results.

7.3 2. Strategy with the best metric

For **KNN**, three strategies have same performance for both weighted-f1 and hamming_loss metrics For **SVM**, Binary Relevance has lowest hamming_loss score and Classifier Chain has highest weighted-f1 score. In general, **Binary Relevance** and **Classifier Chain** have better performance on our models.

7.4 3. If the best method we pick favor any label of size differences

As can be seen in the plots above, macro F1 score is slighly lower than micro F1 score (mean difference 0.1). It indicates that we may fit more poorly in clusters which have fewer movies. We have already tried our best to address the problem of uneven numbers by using stratified k-fold cross-validation. Since the difference between micro and macro F1s is not very large, we can assume all the clusters are correctly classified.