In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import HTML, Math
np.random.seed(1234)
```

4.1 Feature Extraction

In [2]:
```
from sklearn.model_selection import train_test_split
from src.twitter import extract_words, read_vector_file
```

(a) Implement `extract_dictionary(...)` that uses `extract_words(...)` to read all unique words contained in a file into a dictionary. Process the tweets in the order they appear in the file to create this dictionary of d unique words/punctuations.

In [3]:
```
def extract_dictionary(infile):
    """
    Given a filename, reads the text file and builds a dictionary of
    unique words/punctuations.

    Parameters
    ----------
        infile -- string, filename
    Returns
    ----------
        word_list -- dictionary, (key, value) pairs are
                     (word, index)
    """
    with open(infile, 'r') as fid:
        unique_words = set(extract_words(fid.read()))
        return {word: i for i, word in enumerate(unique_words)}
```

(b) Next, implement `extract_feature_vectors(...)` that produces the bag-of-words representation of a file based on the extracted dictionary. That is, for each tweet i, construct a feature vector of length d, where the j^{th} entry in the feature vectors is 1 if the j^{th} word in the dictionary is present in tweet i, or 0 otherwise. For n tweets, save the feature vectors in a feature matrix, where the rows correspond to tweets (examples) and the columns correspond to words (features). Maintain the order of the tweets as they appear in the file.

In [4]:
```
def extract_feature_vectors(infile, word_list):
    """
    Produces a bag-of-words representation of a text file specified
    by the filename infile based on the dictionary word_list.

    Parameters
    ----------
        infile -- string, filename
        world_list -- dictionary, (key, value) pairs are
                      (word, index)

    Returns
    ----------
        feature_matrix -- numpy array of shape (n,d)
                          boolean (0,1) array indicating words
                          presence in a string

    """
    num_lines = sum(1 for line in open(infile, 'r'))
    num_words = len(word_list)
    feature_matrix = np.zeros((num_lines, num_words))

    with open(infile, 'r') as fid:
        for j, tweet in enumerate(fid):
            for word in extract_words(tweet):
                i = word_list[word]
                feature_matrix[j][i] = 1

    return feature_matrix
```

In [5]:
```
data_file = 'data/tweets.txt'
dictionary = extract_dictionary(data_file)
X = extract_feature_vectors(data_file, dictionary)
y = read_vector_file('data/labels.txt')
```

In [6]:
```
pd.options.display.max_rows = 12
pd.options.display.max_columns = 12
pd.DataFrame(X)
```

Out[6]:

	0	1	2	3	4	5	...	1805	1806	1807	1808	1809	1810
0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
624	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
625	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
626	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
627	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
628	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
629	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0

630 rows × 1811 columns

In [7]:
```
with open(data_file, 'r') as f:
    display(next(f))
feature_vec = sorted(zip(X[0], dictionary.keys()), reverse=True)
display(feature_vec[:20], len(feature_vec))
```

```
'2012, Micheal:This Is it && A Christmas Carol Are Cool Movies\n'

[(1.0, 'this'),
 (1.0, 'movies'),
 (1.0, 'micheal'),
 (1.0, 'it'),
 (1.0, 'is'),
 (1.0, 'cool'),
 (1.0, 'christmas'),
 (1.0, 'carol'),
 (1.0, 'are'),
 (1.0, 'a'),
 (1.0, ':'),
 (1.0, '2012'),
 (1.0, ','),
 (1.0, '&'),
 (0.0, '|'),
 (0.0, 'zigster'),
 (0.0, 'z'),
 (0.0, 'yumm'),
 (0.0, 'your'),
 (0.0, 'you')]

1811
```

(c) Split the feature matrix and corresponding labels into your training and test sets. **The first 560 tweets will be used for training and the last 70 tweets will be used for testing.**

In [8]:
```
# note: the data is being shuffled implicitly by train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=
560)
```

```
/usr/lib/python3.7/site-packages/sklearn/model_selection/_split.py:20
69: FutureWarning: From version 0.21, test_size will always complemen
t train_size unless both are specified.
  FutureWarning)
```

(d) Indicate that you have finished the feature extraction and generated the train/test splits in your write-up. ✓

4.2 Hyper-parameter Selection for a Linear-Kernel SVM

In [9]:
```
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
```

(a) The result of a hyperparameter selection often depends upon the choice of performance measure. Here, we will consider the following performance measures: **accuracy**, **F1-score**, and **AUROC**.

Implement `performance(...)` . All measures are implemented in `sklearn.metrics` .

In [10]:
```
def performance(y_true, y_pred, metric="accuracy"):
    """
    Calculates the performance metric based on the agreement between
    the true labels and the predicted labels.

    Parameters
    --------------------
        y_true -- numpy array of shape (n,), known labels
        y_pred -- numpy array of shape (n,), (continuous-valued)
                  predictions
        metric -- string, option used to select the performance
                  measure
                  options: 'accuracy', 'f1-score', 'auroc'

    Returns
    --------------------
        score  -- float, performance score
    """
    # map continuous-valued predictions to binary labels
    y_label = np.sign(y_pred)
    y_label[y_label==0] = 1

    scorer = {
        'accuracy': metrics.accuracy_score,
        'f1_score': metrics.f1_score,
        'auroc': metrics.roc_auc_score,
    }[metric]

    return scorer(y_true, y_label)
```

(b) Next, implement `cv_performance(...)` to return the mean k-fold CV performance for the performance metric passed into the function. Here, you will make use of `SVC.fit(X,y)` and the `SVC.decision_function(X)`, as well as your `performance(...)` function.

You may have noticed that the proportion of the two classes (positive and negative) are not equal in the training data. When dividing the data into folds for CV, you should try to keep the class proportions roughly the same across folds. In your write-up briefly describe why it might be beneficial to maintain class proportions across folds. Then, use `sklearn.cross_validation.StratifiedKFold(...)` to split the data for 5-fold CV, making sure to stratify using only the training labels.

We would want to try to keep class proportions the same across folds to avoid oversampling any one class accidentally. If this were to happen, we might choose a set of examples of all the same class for example. In this scenario, even a trivial classifier that always predicts the majority class can get 100% training accuracy (but will probably have horrible test accuracy because its not a very expressive model). Essentially, it's impossible to learn the class separation when there's an overwhelming majority of a single class, so we use stratified sampling to avoid this situation and keep similar class distributions between the whole data set and any sampled training sets.

In [11]:
```
def cv_performance(clf, X, y, kf, metric="accuracy"):
    """
    Splits the data, X and y, into k-folds and runs k-fold
    cross-validation. Trains classifier on k-1 folds and tests on the
    remaining fold. Calculates the k-fold cross-validation
    performance metric for classifier by averaging the performance
    across folds.

    Parameters
    --------------------
        clf    -- classifier (instance of SVC)
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- cross_validation.KFold or
                  cross_validation.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
        score   -- float, average cross-validation performance across
                   k folds
    """
    scores = []

    for train_index, test_index in kf.split(X, y):
        X_train_fold, X_test_fold = X[train_index], X[test_index]
        y_train_fold, y_test_fold = y[train_index], y[test_index]

        svm = clf.fit(X_train_fold, y_train_fold)
        y_pred = svm.decision_function(X_test_fold)
        scores.append(performance(y_test_fold, y_pred, metric=metric
))

    return sum(scores)/len(scores)
```

In [12]:
```
cv_performance(SVC(gamma='auto'), X_train, y_train, StratifiedKFold(n
_splits=5), metric='accuracy')
```

Out[12]: 0.6535764114303937

(c) Now, implement `select_param_linear(...)` to choose a setting for C for a linear SVM based on the training data and the specified metric. Your function should call `cv_performance(...)`, passing in instances of `SVC(kernel='linear', C=c)` with different values for `C`, e.g. $C = 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 10^2$

In [13]:
```
def select_param_linear_helper(c_range, X, y, kf, metric='accuracy'):
    return {
        c:
        cv_performance(
            SVC(kernel='linear', C=c, gamma='auto'),
            X, y, kf, metric=metric
        )
        for c in c_range
    }
```

In [14]:
```
def select_param_linear(X, y, kf, metric="accuracy"):
    """
    Sweeps different settings for the hyperparameter of a linear-
    kernel SVM, calculating the k-fold CV performance for each
    setting, then selecting the hyperparameter that 'maximize'
    the average k-fold CV performance.

    Parameters
    --------------------
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- cross_validation.KFold or
                  cross_validation.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
        C -- float, optimal parameter value for linear-kernel SVM
    """

    print('Linear SVM Hyperparameter Selection based on ' + str(metri
c) + ':')
    c_range = 10.0 ** np.arange(-3, 3)
    results = select_param_linear_helper(c_range, X, y, kf, c_range,
metric=metric)
    return c_range[np.argmax(results)]
```

(d) Finally, using the training data from Section 4.1 and the functions implemented here, find the best setting for C for each performance measure mentioned above. Report your findings in tabular format (up to the fourth decimal place).

In [15]:
```
c_range = 10.0 ** np.arange(-3, 3)
kf = StratifiedKFold(n_splits=5)
scores = {
    metric: select_param_linear_helper(
        c_range,
        X_train,
        y_train,
        kf,
        metric=metric,
    )
    for metric in ['accuracy', 'f1_score', 'auroc']
}
```

In [16]:
```
df = pd.DataFrame(data=scores)
df = df.rename_axis('C', axis='columns')
df.append(df.idxmax().rename('best C')).round(4)
```

Out[16]:

C	accuracy	f1_score	auroc
0.001	0.6536	0.7905	0.5000
0.01	0.7822	0.8497	0.7109
0.1	0.8072	0.8584	0.7688
1.0	0.8250	0.8675	0.8019
10.0	0.8196	0.8636	0.7953
100.0	0.8196	0.8636	0.7953
best C	1.0000	1.0000	1.0000

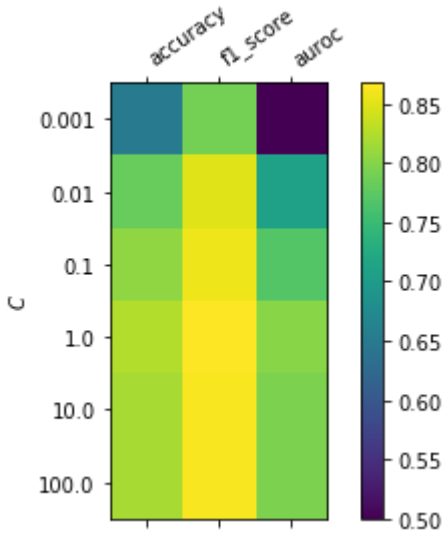
Your `select_param_linear(...)` function returns the 'best' C given a range of values. How does the 5-fold CV performance vary with C and the performance metric?

In general, it seems that AUROC is the most pessimistic metric of the three, F1 is optimistic, and accuracy is middle-of-the-road.

For lower values of C, i.e. $C < 1$, the accuracy of the SVM across all performance metrics grows worse as C decreases. For $C > 1$, we note that the performance seems to level off across all performance metrics as the SVM becomes less willing to provide slack and penalizes misclassified examples more harshly. Performance peaks at $C = 1$ for all metrics because $C = 1$ is the choice for this dataset where enough slack is afforded to account for any outlier examples while still maintaining a hard enough margin to accurately separate all other data.

In [17]:
```
fig, ax = plt.subplots(ncols=1)

ax.set_ylabel('$C$')
ax.set_xticklabels(['', 'accuracy', 'f1_score', 'auroc'], rotation=35
, ha='left')
ax.set_yticklabels([''] + list(map(str, c_range)))
im = ax.matshow(df)
fig.colorbar(im, ax=ax);
```



4.3 Test Set Performance

(a) Based on the results you obtained in Section 4.2, choose a hyperparameter setting for the linear-kernel SVM. Then, using the training data extracted in Section 4.1 and `SVC.fit(...)`, train a linear-kernel SVM with your chosen settings.

In [18]:
```
clf = SVC(kernel='linear', gamma='auto', C=1.0)
clf.fit(X_train, y_train)
```

Out[18]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='line
ar',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

(b) Implement `performance_test(...)` which returns the value of a performance measure, given the test data and a trained classifier.

In [19]:
```
def performance_test(clf, X, y, metric="accuracy"):
    """
    Estimates the performance of the classifier using the 95% CI.

    Parameters
    ----------
        clf    -- classifier (instance of SVC) [already fit to data]
        X      -- numpy array of shape (n,d), feature vectors of test
                  set
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1} of
                  test set
        metric -- string, option used to select performance measure

    Returns
    ----------
        score  -- float, classifier performance
    """

    y_pred = clf.decision_function(X)
    return performance(y, y_pred, metric=metric)
```

(c) For each performance metric, use `performance_test(...)` and the trained linear-kernel SVM classifier to measure performance on the test data. Report the results. Be sure to include the name of the performance metric employed and the performance on the test data.

In [20]:
```
performance_scores = {
    metric: performance_test(clf, X_test, y_test, metric=metric)
    for metric in ['accuracy', 'f1_score', 'auroc']
}
```

In [21]:
```
pd.DataFrame(performance_scores, index=['performance:'])
```

Out[21]:

	accuracy	f1_score	auroc
performance:	0.857143	0.9	0.849359

NOTE: training data has been shuffled

In [ ]:
```

```