НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Дисциплина: «Анализ данных»

Домашнее задание на тему:

«Лабораторная работа №6»

Выполнил: Осипов Лев,

студент группы 301ПИ (1).

Москва, 2015 г.

**СОДЕРЖАНИЕ**

**Теоретическая часть3**

**Задание 13**

**Задание 23**

**Задание 33**

**Практическая часть4**

**Список литературы9**

**Текст программы10**

**ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

**Задание 1**

Разделяющая поверхность при проекции на исходное пространство будет иметь форму эллипса, дуги эллипса или же прямой. Представим, что объекты разных классов в пространстве расположены на двух разных дугах одного эллипса и равноудалены друг от друга. В таком случае проекция разделяющей плоскости будет дугой, которая проходит между объектами классов.

**Задание 2**

1. **Линейный SVM**

У нас есть точки в d пространствах и необходимо разделить их плоскостью. После обучения мы имеем разделяющую гиперплоскость в d пространствах, описываемую d элементами. Каждый объект так же описывается d признаками, а для его классификации нам необходимо вычислить его местоположение относительно гиперплоскости. Так как количество признаков d, нам потребуется **O(d)** операций для классификации.

1. **Ядерный SVM**

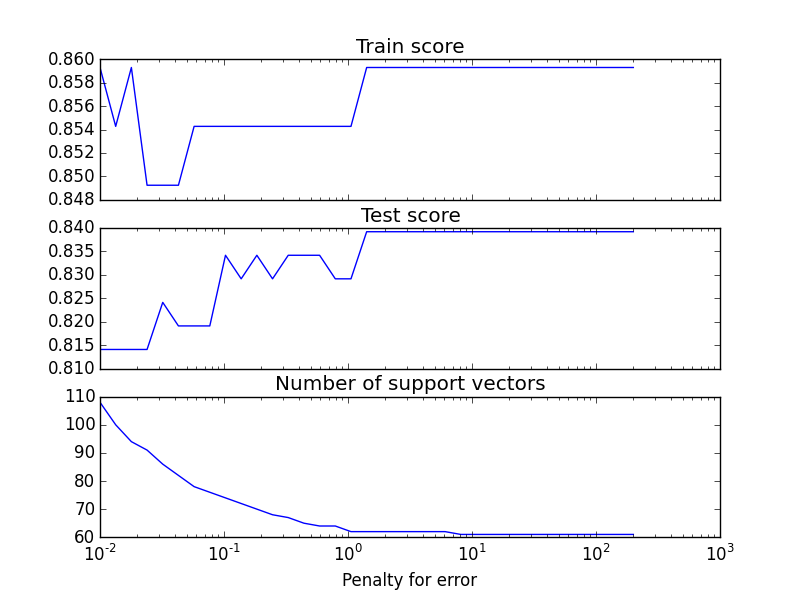
В случае же ядерного SVM некоторые ядра могут иметь размерности, стремящиеся к бесконечности. Можно пойти по-другому: так как алгоритм использует только опорные вектора (формула линейного классификатора из презентации), возьмем их количество за N. Также присутствует зависимость от некой функции ядра сложностью K. Поэтому итоговая сложность будет **O(N)\*O(K)**.

**Задание 3**

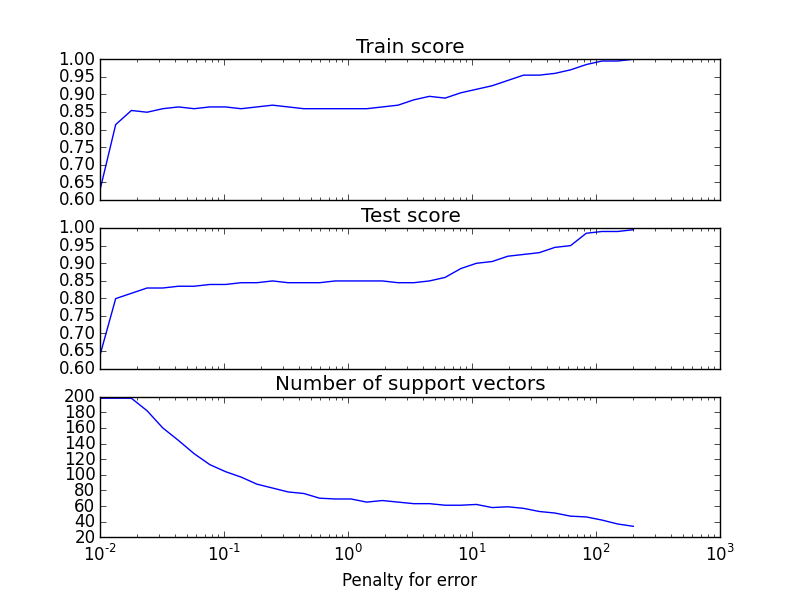
Сложение ядер происходит на основе конкатенации векторов. Поэтому мы сможем линейно разделить выборку, если проведем плоскость в пространстве первого ядра с учетом конкатенации векторов обоих ядер.

**ПРАКТИЧЕСКАЯ ЧАСТЬ**

Для решения задания была написана программа, исследующая зависимость работы алгоритма от решающих правил для линейного и RBF ядер.

****

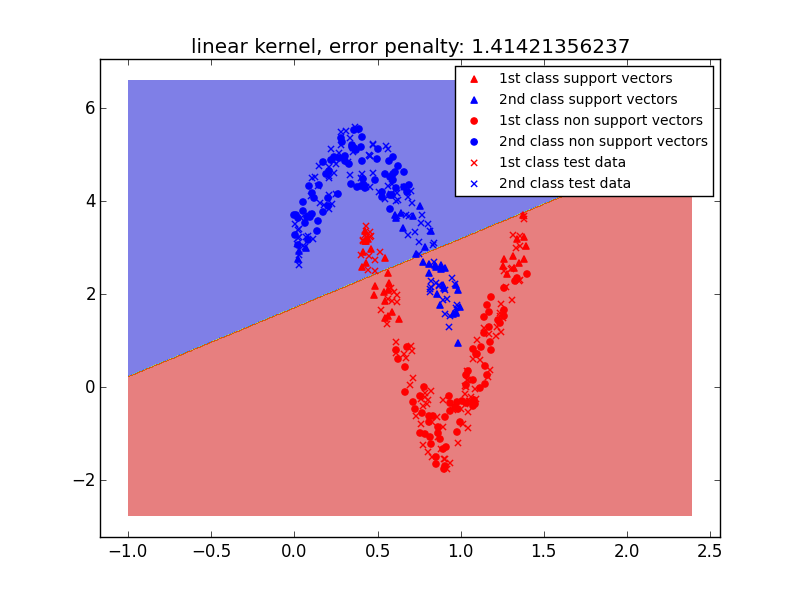
*Рис. 1. Зависимость верно классифицированных объектов на обучающей и тестовой выборках, а также количество опорных векторов от С. Линейное ядро*

****

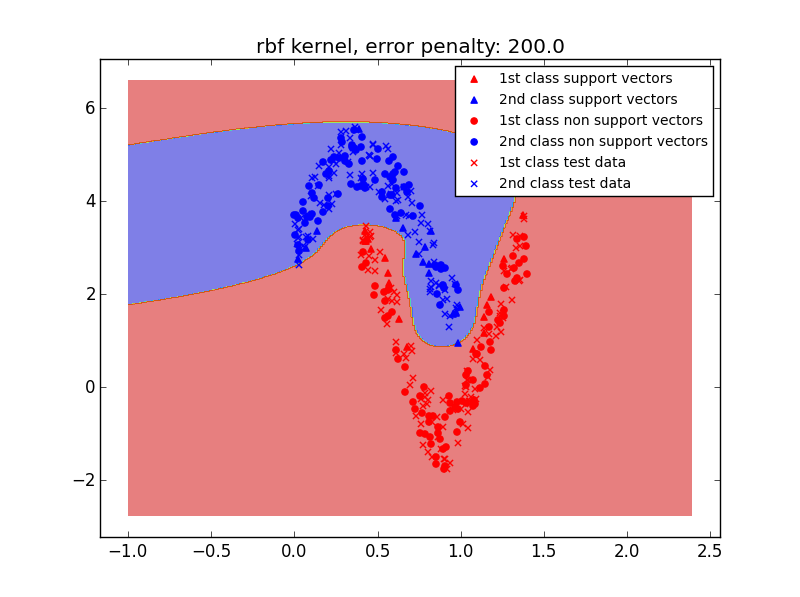
*Рис. 2. Зависимость верно классифицированных объектов на обучающей и тестовой выборках, а также количество опорных векторов от С. RBF ядро*

В обоих ядрах наблюдается снижение количества опорных векторов, однако в случае RBF ядра снижение происходит медленнее.

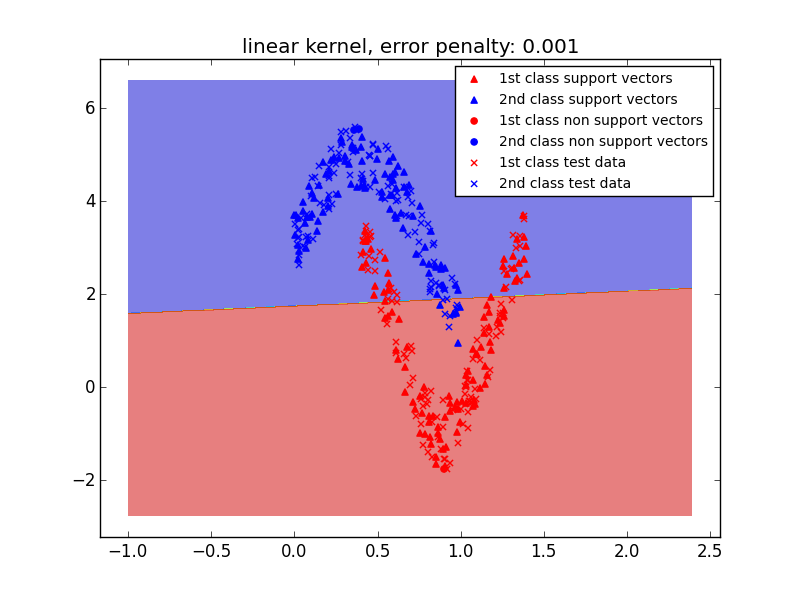
Также в обоих ядрах наблюдается возрастание точности классификации, однако в случае RBF ядра на обучающей и тестовой выборках динамика возрастания весьма схожа, в отличие от случая линейного ядра.



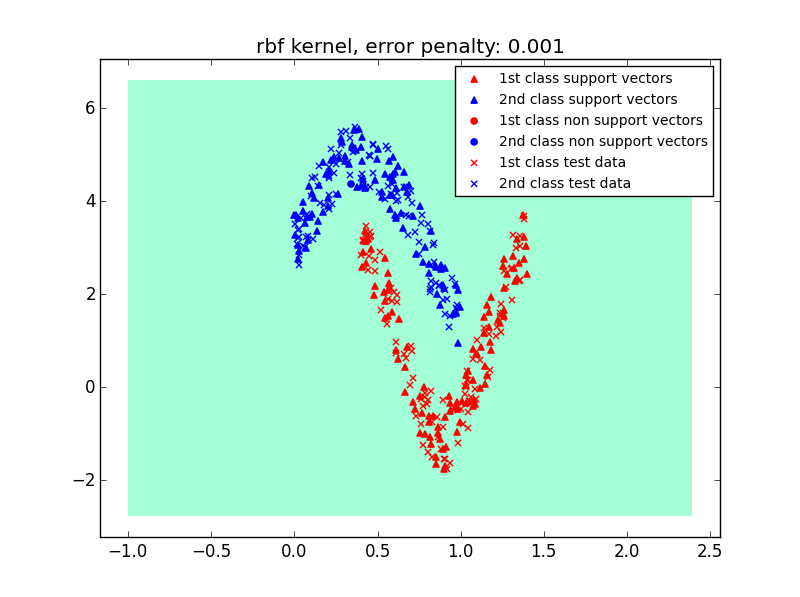
*Рис. 3. Оптимальное С. Линейное ядро*



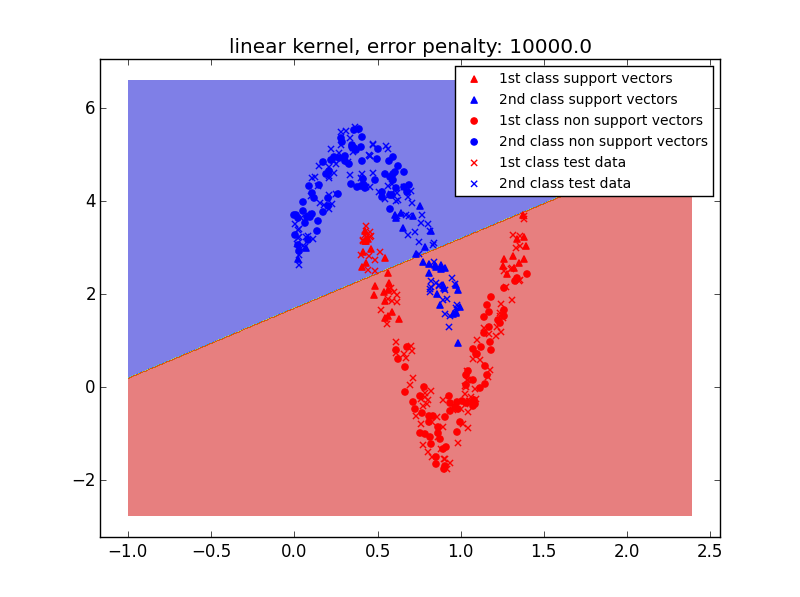
*Рис. 4. Оптимальное С. RBF ядро*



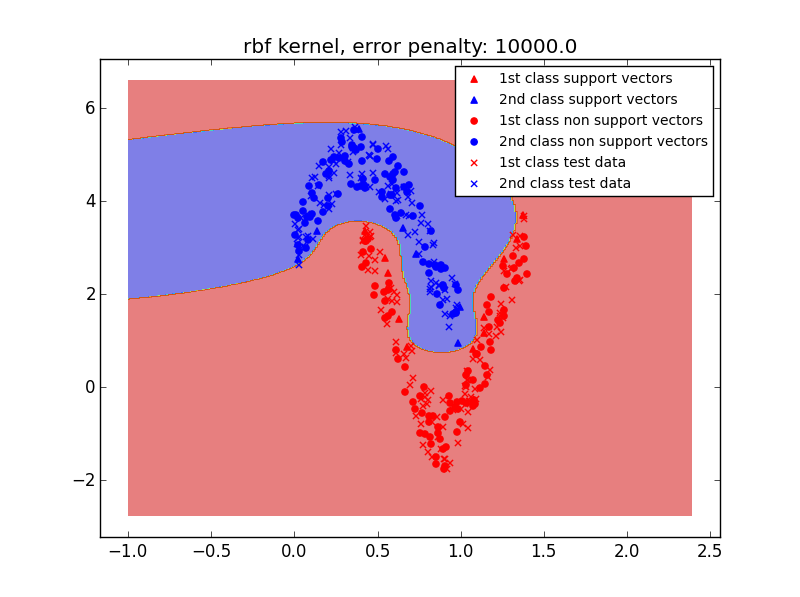
*Рис. 5. Неадекватно маленькое С. Линейное ядро*



*Рис. 6. Неадекватно маленькое С. RBF ядро*

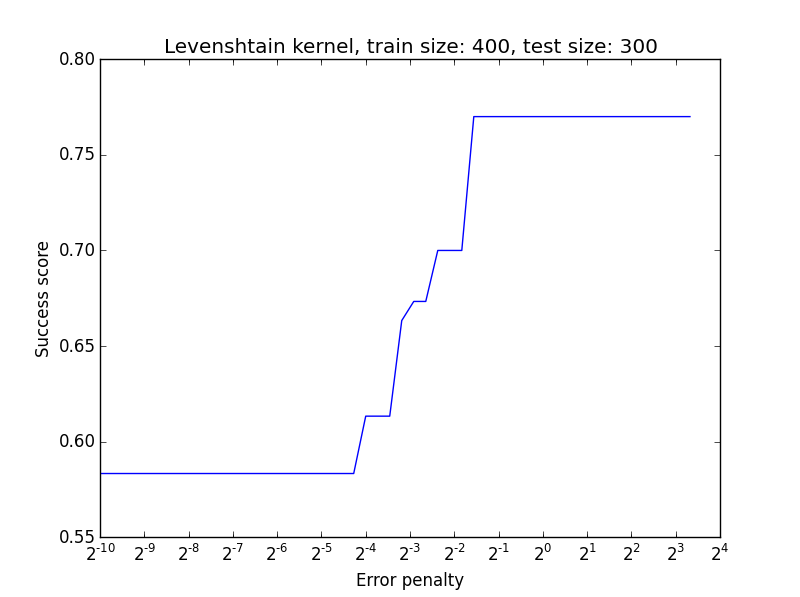
****

*Рис. 7. Неадекватно большое С. Линейное ядро*

****

*Рис. 8. Неадекватно большое С. RBF ядро*

Видно, что в обоих ядрах самая неудачная классификация происходит при условии неадекватно маленького С. В RBF ядре этот случай критичен - все объекты были помечены одним классом.Также был обучено собственное ядро, определяющее близость между строками (для решения задачи классификации слов). Удалось добиться точности 0.77 на тестовой выборке.



*Рис. 8. Точность классификации на тестовой выборке. «Ядро Левенштейна»*

**СПИСОК ЛИТЕРАТУРЫ**

1. **Анализ данных (Программная инженерия)** –   
   http://wiki.cs.hse.ru/Анализ\_данных\_(Программная\_инженерия)

**ТЕКСТ ПРОГРАММЫ**

\_\_author\_\_ = 'Lev Osipov'  
  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** sklearn.svm **import** SVC  
**from** math **import** log10  
**from** math **import** log  
**from** Levenshtein **import** distance  
  
  
**def learn\_stat**(train\_data, train\_labels, test\_data, test\_labels, kernel, C):  
 clf = SVC(C=C, kernel=kernel)  
 clf.fit(train\_data, train\_labels)  
 train\_score = clf.score(train\_data, train\_labels)  
 test\_score = clf.score(test\_data, test\_labels)  
 support = clf.support\_.size  
 **return** train\_score, test\_score, support  
  
  
**def plot\_stat**(train\_data, train\_labels, test\_data, test\_labels, kernel, minC, maxC, steps):  
  
 train\_score = np.empty(steps)  
 test\_score = np.empty(steps)  
 support = np.empty(steps)  
  
 c\_interval = np.logspace(minC, maxC, steps)  
 **for** i, C **in** enumerate(c\_interval):  
 train\_score[i], test\_score[i], support[i] = \  
 learn\_stat(train\_data, train\_labels, test\_data, test\_labels, kernel, C)  
 f, ax = plt.subplots(3, sharex=True)  
 i = 0  
  
 ax[i].plot(c\_interval, train\_score)  
 ax[i].set\_title("Train score")  
 ax[i].set\_xscale('log', basex=10)  
 i += 1  
  
 ax[i].plot(c\_interval, test\_score)  
 ax[i].set\_title("Test score")  
 ax[i].set\_xscale('log', basex=10)  
 i += 1  
  
 ax[i].plot(c\_interval, support)  
 ax[i].set\_title("Number of support vectors")  
 ax[i].set\_xlabel("Penalty for error")  
  
 ax[i].set\_xscale('log', basex=10)  
  
 plt.show()  
  
  
**def plot\_support**(train\_data, train\_labels, test\_data, test\_labels, kernel, C):  
  
 clf = SVC(C=C, kernel=kernel)  
 clf.fit(train\_data, train\_labels)  
  
 support\_i = clf.support\_  
  
 train\_sup\_vec = train\_data[support\_i]  
 train\_sup\_lab = train\_labels[support\_i]  
  
 train\_non\_sup\_vec = np.delete(train\_data, support\_i, axis=0)  
 train\_non\_sup\_lab = np.delete(train\_labels, support\_i, axis=0)  
  
 train\_sup\_vec\_1 = train\_sup\_vec[np.where(train\_sup\_lab == 1)]  
 train\_sup\_vec\_2 = train\_sup\_vec[np.where(train\_sup\_lab != 1)]  
  
 train\_non\_sup\_vec\_1 = train\_non\_sup\_vec[np.where(train\_non\_sup\_lab == 1)]  
 train\_non\_sup\_vec\_2 = train\_non\_sup\_vec[np.where(train\_non\_sup\_lab != 1)]  
  
 test\_data\_1 = test\_data[np.where(test\_labels == 1)]  
 test\_data\_2 = test\_data[np.where(test\_labels != 1)]  
  
 all = np.concatenate((test\_data, train\_data), axis=0)  
 h = .01  
  
 x\_min, x\_max = all[:, 0].min() - 1, all[:, 0].max() + 1  
 y\_min, y\_max = all[:, 1].min() - 1, all[:, 1].max() + 1  
 xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),  
 np.arange(y\_min, y\_max, h))  
  
 z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])  
 z = z.reshape(xx.shape)  
 plt.contourf(xx, yy, z, alpha=0.5)  
  
 plt.scatter(train\_sup\_vec\_1[:, 0], train\_sup\_vec\_1[:, 1], color='r', marker='^',  
 label='1st class support vectors')  
 plt.scatter(train\_sup\_vec\_2[:, 0], train\_sup\_vec\_2[:, 1], color='b', marker='^',  
 label='2nd class support vectors')  
  
 plt.scatter(train\_non\_sup\_vec\_1[:, 0], train\_non\_sup\_vec\_1[:, 1], color='r', marker='o',  
 label='1st class non support vectors')  
 plt.scatter(train\_non\_sup\_vec\_2[:, 0], train\_non\_sup\_vec\_2[:, 1], color='b', marker='o',  
 label='2nd class non support vectors')  
  
 plt.scatter(test\_data\_1[:, 0], test\_data\_1[:, 1], color='r', marker='x', label='1st class test data')  
 plt.scatter(test\_data\_2[:, 0], test\_data\_2[:, 1], color='b', marker='x', label='2nd class test data')  
  
 plt.legend(scatterpoints=1, fontsize=10)  
 plt.title("{0} kernel, error penalty: {1}".format(kernel, C))  
  
 plt.show()  
 plt.clf()  
  
  
**def find\_optimal\_c**(train\_data, train\_labels, test\_data, test\_labels, kernel, minC, maxC, steps):  
 c\_interval = np.logspace(minC, maxC, steps)  
 max\_score = 0  
 c = minC  
 **for** i, C **in** enumerate(c\_interval):  
 test\_score = learn\_stat(train\_data, train\_labels, test\_data, test\_labels, kernel, C)[1]  
 **if** test\_score > max\_score:  
 max\_score = test\_score  
 c = C  
 **return** c  
  
# Task 1  
tr\_data = pd.read\_csv('synth\_train.csv').as\_matrix()  
tr\_labels = tr\_data[:, 0]  
tr\_data = np.delete(tr\_data, 0, axis=1)  
  
te\_data = pd.read\_csv('synth\_test.csv').as\_matrix()  
te\_labels = te\_data[:, 0]  
te\_data = np.delete(te\_data, 0, axis=1)  
  
# CHANGE 'rbf' to 'linear' to see linear kernel results  
  
# Task 2  
minimC = log10(1e-2)  
maximC = log10(200)  
steps\_count = 35  
plot\_stat(tr\_data, tr\_labels, te\_data, te\_labels, 'rbf', minimC, maximC, steps\_count)  
  
# Task 3  
c = find\_optimal\_c(tr\_data, tr\_labels, te\_data, te\_labels, 'rbf', minimC, maximC, steps\_count)  
plot\_support(tr\_data, tr\_labels, te\_data, te\_labels, 'rbf', 1e-3)  
plot\_support(tr\_data, tr\_labels, te\_data, te\_labels, 'rbf', c)  
plot\_support(tr\_data, tr\_labels, te\_data, te\_labels, 'rbf', 1e4)  
  
# Levenshtein  
  
  
**def levenshtein\_gram\_matrix**(s1, s2):  
 res = np.empty((len(s1), len(s2)))  
 **for** i **in** range(len(s1)):  
 **for** j **in** range(len(s2)):  
 res[i][j] = 1 / (1 + distance(s1[i], s2[j]))  
 **return** res  
  
**with** open('en.txt') **as** f:  
 en = f.readlines()  
  
**with** open('fr.txt') **as** f:  
 fr = f.readlines()  
  
data = np.array(en + fr)  
  
labels = np.concatenate((np.ones(len(en)), -1 \* np.ones(len(fr))))  
data = np.column\_stack((data, labels))  
np.random.shuffle(data)  
  
tr\_length = 400  
tr\_data = data[:tr\_length]  
tr\_labels = tr\_data[:, 1]  
tr\_data = tr\_data[:, 0]  
  
te\_length = 300  
te\_data = data[tr\_length:tr\_length+te\_length]  
te\_labels = te\_data[:, 1]  
te\_data = te\_data[:, 0]  
  
train\_gram\_matrix = levenshtein\_gram\_matrix(tr\_data, tr\_data)  
test\_gram\_matrix = levenshtein\_gram\_matrix(te\_data, tr\_data)  
  
c\_range = np.logspace(log(0.001, 2), log(10, 2), base=2)  
scores = []  
**for** C **in** c\_range:  
 clf = SVC(C=C, kernel='precomputed')  
 clf.fit(train\_gram\_matrix, tr\_labels)  
 scores.append(clf.score(test\_gram\_matrix, te\_labels))  
  
**print**("Best: {0}".format(max(scores)))  
  
plt.plot(c\_range, scores)  
plt.xscale('log', basex=2)  
plt.xlabel("Error penalty")  
plt.ylabel("Success score")  
plt.title("Levenshtain kernel, train size: {0}, test size: {1}".format(tr\_length, te\_length))  
plt.show()