Предобработка данных и логистическая регрессия для задачи бинарной классификации

В задании вам будет предложено ознакомиться с основными техниками предобработки данных, а так же применить их для обучения модели логистической регрессии. Ответ потребуется загрузить в соответствующую форму в виде 6 текстовых файлов.

Для выполнения задания требуется Python версии 2.7, а также актуальные версии библиотек:

- NumPy: 1.10.4 и выше
- Pandas: 0.17.1 и выше
- Scikit-learn: 0.17 и выше

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
matplotlib.style.use('ggplot')
%matplotlib inline
```

Описание датасета

Задача: по 38 признакам, связанных с заявкой на грант (область исследований учёных, информация по их академическому бэкграунду, размер гранта, область, в которой он выдаётся) предсказать, будет ли заявка принята. Датасет включает в себя информацию по 6000 заявкам на гранты, которые были поданы в университете Мельбурна в период с 2004 по 2008 год.

Полную версию данных с большим количеством признаков можно найти на https://www.kaggle.com/c/unimelb.

In [2]:
```
data = pd.read_csv('data.csv')
data.shape
```

Out[2]:
```
(6000, 39)
```

Выделим из датасета целевую переменную Grant.Status и обозначим её за y Теперь X обозначает обучающую выборку, y - ответы на ней

In [3]:
```
X = data.drop('Grant.Status', 1)
y = data['Grant.Status']
```

Теория по логистической регрессии

После осознания того, какую именно задачу требуется решить на этих данных, следующим шагом при реальном анализе был бы подбор подходящего метода. В данном задании выбор метода было произведён за вас, это логистическая регрессия. Кратко напомним вам используемую модель.

Логистическая регрессия предсказывает вероятности принадлежности объекта к каждому классу. Сумма ответов логистической регрессии на одном объекте для всех классов равна единице.

$$\sum_{k=1}^{K} \pi_{ik} = 1, \quad \pi_k \equiv P\left(y_i = k \mid x_i, \theta\right),$$

где:

- π_{ik} - вероятность принадлежности объекта x_i из выборки X к классу k
- θ - внутренние параметры алгоритма, которые настраиваются в процессе обучения, в случае логистической регрессии - w, b

Из этого свойства модели в случае бинарной классификации требуется вычислить лишь вероятность принадлежности объекта к одному из классов (вторая вычисляется из условия нормировки вероятностей). Эта вероятность вычисляется, используя логистическую функцию:

$$P\left(y_i = 1 \mid x_i, \theta\right) = \frac{1}{1 + \exp(-w^T x_i - b)}$$

Параметры w и b находятся, как решения следующей задачи оптимизации (указаны функционалы с L1 и L2 регуляризацией, с которыми вы познакомились в предыдущих заданиях):

L2-regularization:

$$Q(X, y, \theta) = \frac{1}{2} w^T w + C \sum_{i=1}^{l} \log(1 + \exp(-y_i(w^T x_i + b))) \longrightarrow \min_{w,b}$$

L1-regularization:

$$Q(X, y, \theta) = \sum_{d=1}^{D} |w_d| + C \sum_{i=1}^{l} \log(1 + \exp(-y_i(w^T x_i + b))) \longrightarrow \min_{w,b}$$

C - это стандартный гиперпараметр модели, который регулирует то, насколько сильно мы позволяем модели подстраиваться под данные.

Предобработка данных

Из свойств данной модели следует, что:

- все X должны быть числовыми данными (в случае наличия среди них категорий, их требуется некоторым способом преобразовать в вещественные числа)
- среди X не должно быть пропущенных значений (т.е. все пропущенные значения перед применением модели следует каким-то образом заполнить)

Поэтому базовым этапом в предобработке любого датасета для логистической регрессии будет кодирование категориальных признаков, а так же удаление или интерпретация пропущенных значений (при наличии того или другого).

In [4]:
```
data.head()
```

Out[4]:

	Grant.Status	Sponsor.Code	Grant.Category.Code	Contract.Value.Band...see.note.A	RFCD.Code.1	RFCD.Percenta
0	1	21A	50A	A	230202.0	
1	1	4D	10A	D	320801.0	
2	0	NaN	NaN	NaN	320602.0	
3	0	51C	20C	A	291503.0	
4	0	24D	30B	NaN	380107.0	

5 rows × 39 columns

Видно, что в датасете есть как числовые, так и категориальные признаки. Получим списки их названий:

In [5]:
```
numeric_cols = ['RFCD.Percentage.1', 'RFCD.Percentage.2', 'RFCD.Percentage.3',
                'RFCD.Percentage.4', 'RFCD.Percentage.5',
                'SEO.Percentage.1', 'SEO.Percentage.2', 'SEO.Percentage.3',
                'SEO.Percentage.4', 'SEO.Percentage.5',
                'Year.of.Birth.1', 'Number.of.Successful.Grant.1', 'Number.of.Unsuccessful.Grar
categorical_cols = list(set(X.columns.values.tolist()) - set(numeric_cols))
```

Также в нём присутствуют пропущенные значения. Очевидны решением будет исключение всех данных, у которых пропущено хотя бы одно значение. Сделаем это:

In [6]:
```
data.dropna().shape
```

Out[6]:
```
(213, 39)
```

Видно, что тогда мы выбросим почти все данные, и такой метод решения в данном случае не сработает.

Пропущенные значения можно так же интерпретировать, для этого существует несколько способов, они различаются для категориальных и вещественных признаков.

Для вещественных признаков:

- заменить на 0 (данный признак давать вклад в предсказание для данного объекта не будет)
- заменить на среднее (каждый пропущенный признак будет давать такой же вклад, как и среднее значение признака на датасете)

Для категориальных:

- интерпретировать пропущенное значение, как ещё одну категорию (данный способ является самым естественным, так как в случае категорий у нас есть уникальная возможность не потерять информацию о наличии пропущенных значений; обратите внимание, что в случае вещественных признаков данная информация неизбежно теряется)

Задание 0. Обработка пропущенных значений.

1. Заполните пропущенные вещественные значения в X нулями и средними по столбцам, назовите полученные датафреймы X_real_zeros и X_real_mean соответственно. Для подсчёта средних используйте описанную ниже функцию calculate_means, которой требуется передать на вход вещественные признаки из исходного датафрейма.
2. Все категориальные признаки в X преобразуйте в строки, пропущенные значения требуется также преобразовать в какие-либо строки, которые не являются категориями (например, 'NA'), полученный датафрейм назовите X_cat.

Для объединения выборок здесь и далее в задании рекомендуется использовать функции

```
np.hstack(...)
np.vstack(...)
```

In [7]:
```
def calculate_means(numeric_data):
    means = np.zeros(numeric_data.shape[1])
    for j in range(numeric_data.shape[1]):
        to_sum = numeric_data.iloc[:,j]
        indices = np.nonzero(~numeric_data.iloc[:,j].isnull())[0]
        correction = np.amax(to_sum[indices])
        to_sum /= correction
        for i in indices:
            means[j] += to_sum[i]
```

```
        means[j] /= indices.size
        means[j] *= correction
    return pd.Series(means, numeric_data.columns)
```

In [8]:
```
# place your code here
X_real_zeros = X[numeric_cols].fillna(0)
```

In [9]:
```
means = calculate_means(X[numeric_cols])
X_real_mean = X[numeric_cols]
for i in range(len(numeric_cols)):
    X_real_mean.iloc[:, i] = X_real_mean.iloc[:, i].fillna(means.values[i])
```

```
C:\Anaconda2\lib\site-packages\pandas\core\indexing.py:545: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
#indexing-view-versus-copy
  self.obj[item_labels[indexer[info_axis]]] = value
```

In [10]:
```
X_cat = X[categorical_cols]
X_cat = X_cat.fillna('NA')
X_cat = X_cat.astype(str)
```

Преобразование категориальных признаков.

В предыдущей ячейке мы разделили наш датасет ещё на две части: в одной присутствуют только вещественные признаки, в другой только категориальные. Это понадобится нам для раздельной последующей обработке этих данных, а так же для сравнения качества работы тех или иных методов.

Для использования модели регрессии требуется преобразовать категориальные признаки в вещественные. Рассмотрим основной способ преоборазования категориальных признаков в вещественные: one-hot encoding. Его идея заключается в том, что мы преобразуем категориальный признак при помощи бинарного кода: каждой категории ставим в соответствие набор из нулей и единиц.

Посмотрим, как данный метод работает на простом наборе данных.

In [11]:
```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.feature_extraction import DictVectorizer as DV

categorial_data = pd.DataFrame({'sex': ['male', 'female', 'male', 'female'],
                                'nationality': ['American', 'European', 'Asian', 'European']})
print('Исходные данные:\n')
print(categorial_data)
encoder = DV(sparse = False)
encoded_data = encoder.fit_transform(categorial_data.T.to_dict().values())
print('\nЗакодированные данные:\n')
print(encoded_data)
```

```
Исходные данные:

  nationality     sex
0    American    male
1    European  female
2       Asian    male
3    European  female

Закодированные данные:

[[ 1.  0.  0.  0.  1.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  1.  0.  0.  1.]
 [ 0.  0.  1.  1.  0.]]
```

Как видно, в первые три колонки оказалась закодированна информация о стране, а во вторые две - о поле. При этом для совпадающих элементов выборки строки будут полностью совпадать. Также из примера видно, что кодирование признаков сильно увеличивает их количество, но полностью сохраняет информацию, в том числе о наличии пропущенных значений (их наличие просто становится одним из бинарных признаков в преобразованных данных).

Теперь применим one-hot encoding к категориальным признакам из исходного датасета. Обратите внимание на общий для всех методов преобработки данных интерфейс. Функция

```
encoder.fit_transform(X)
```

позволяет вычислить необходимые параметры преобразования, впоследствии к новым данным можно уже применять функцию

```
encoder.transform(X)
```

Очень важно применять одинаковое преобразование как к обучающим, так и тестовым данным, потому что в противном случае вы получите непредсказуемые, и, скорее всего, плохие результаты. В частности, если вы отдельно закодируете обучающую и тестовую выборку, то получите вообще говоря разные коды для одних и тех же признаков, и ваше решение работать не будет.

Также параметры многих преобразований (например, рассмотренное ниже масштабирование) нельзя вычислять одновременно на данных из обучения и теста, потому что иначе подсчитанные на тесте метрики качества будут давать смещённые оценки на качество работы алгоритма. Кодирование категориальных признаков не считает на обучающей выборке никаких параметров, поэтому его можно применять сразу к всему датасету.

In [12]:
```
encoder = DV(sparse = False)
X_cat_oh = encoder.fit_transform(X_cat.T.to_dict().values())
```

Для построения метрики качества по результату обучения требуется разделить исходный датасет на обучающую и тестовую выборки.

Обращаем внимание на заданный параметр для генератора случайных чисел: random_state. Так как результаты на обучении и тесте будут зависеть от того, как именно вы разделите объекты, то предлагается использовать заранее определённое значение для получение результатов, согласованных с ответами в системе проверки заданий.

In [13]:
```
X_cat_oh = pd.DataFrame(X_cat_oh)
from sklearn.cross_validation import train_test_split

(X_train_real_zeros,
 X_test_real_zeros,
 y_train, y_test) = train_test_split(X_real_zeros, y,
                                     test_size=0.3,
                                     random_state=0)
(X_train_real_mean,
 X_test_real_mean) = train_test_split(X_real_mean,
                                      test_size=0.3,
                                      random_state=0)
(X_train_cat_oh,
 X_test_cat_oh) = train_test_split(X_cat_oh,
                                   test_size=0.3,
                                   random_state=0)
```

In [56]:
```
y_test
```

Out[56]:
```
3068    0
3835    0
4872    0
2009    0
5197    0
2912    0
5988    1
2584    0
3776    1
4779    0
2736    0
3149    1
1451    1
4224    1
224     0
399     0
4039    0
3472    0
3862    0
235     0
3221    1
5231    1
2438    1
383     1
2075    0
5984    0
2039    0
148     0
5049    0
3740    0
       ..
1517    1
3361    1
3597    1
3556    0
529     1
2971    0
3532    0
2225    0
2853    1
567     0
4676    0
2077    0
4436    0
1872    0
1740    0
3493    0
1703    0
2570    0
4375    1
3894    0
3800    1
2142    0
3335    0
48      0
1136    0
496     1
1653    1
5175    1
4236    0
4483    0
Name: Grant.Status, dtype: int64
```

Описание классов

Итак, мы получили первые наборы данных, для которых выполнены оба ограничения логистической регрессии на входные данные. Обучим на них регрессию, используя имеющийся в библиотеке sklearn функционал по подбору гиперпараметров модели

```
optimizer = GridSearchCV(estimator, param_grid)
```

где:

- estimator - обучающий алгоритм, для которого будет производиться подбор параметров
- param_grid - словарь параметров, ключами которого являются строки-названия, которые передаются алгоритму estimator, а значения - набор параметров для перебора

Данный класс выполняет кросс-валидацию обучающей выборки для каждого набора параметров и находит те, на которых алгоритм работает лучше всего. Этот метод позволяет настраивать гиперпараметры по обучающей выборке, избегая переобучения. Некоторые опциональные параметры вызова данного класса, которые нам понадобятся:

- scoring - функционал качества, максимум которого ищется кросс валидацией, по умолчанию используется функция score() класса esimator
- n_jobs - позволяет ускорить кросс-валидацию, выполняя её параллельно, число определяет количество одновременно запущенных задач
- cv - количество фолдов, на которые разбивается выборка при кросс-валидации

После инициализации класса GridSearchCV, процесс подбора параметров запускается следующим методом:

```
optimizer.fit(X, y)
```

На выходе для получения предсказаний можно пользоваться функцией

```
optimizer.predict(X)
```

для меток или

```
optimizer.predict_proba(X)
```

для вероятностей (в случае использования логистической регрессии).

Также можно напрямую получить оптимальный класс estimator и оптимальные параметры, так как они является атрибутами класса GridSearchCV:

- best_estimator_ - лучший алгоритм
- best_params_ - лучший набор параметров

Класс логистической регрессии выглядит следующим образом:

```
estimator = LogisticRegression(penalty)
```

где penalty принимает либо значение 'l2', либо 'l1'. По умолчанию устанавливается значение 'l2', и везде в задании, если об этом не оговорено особо, предполагается использование логистической регрессии с L2-регуляризацией.

Задание 1. Сравнение способов заполнения вещественных пропущенных значений.

1. Составьте две обучающие выборки из вещественных и категориальных признаков: в одной вещественные признаки, где пропущенные значения заполнены нулями, в другой - средними. Рекомендуется записывать в выборки сначала вещественные, а потом категориальные признаки.
2. Обучите на них логистическую регрессию, подбирая параметры из заданной сетки param_grid по методу кросс-валидации с числом фолдов cv=3. В качестве оптимизируемой функции используйте заданную по умолчанию.
3. Постройте два графика оценок точности +- их стандратного отклонения в зависимости от гиперпараметра и убедитесь, что вы действительно нашли её максимум. Также обратите внимание на большую дисперсию получаемых оценок (уменьшить её можно увеличением числа фолдов cv).
4. Получите две метрики качества AUC ROC на тестовой выборке и сравните их между собой. Какой способ заполнения пропущенных вещественных значений работает лучше? В дальнейшем для выполнения задания в качестве вещественных признаков используйте ту выборку, которая даёт лучшее качество на тесте.
5. Передайте два значения AUC ROC (сначала для выборки, заполненной средними, потом для выборки, заполненной нулями) в функцию write_answer_1 и запустите её. Полученный файл является ответом на 1 задание.

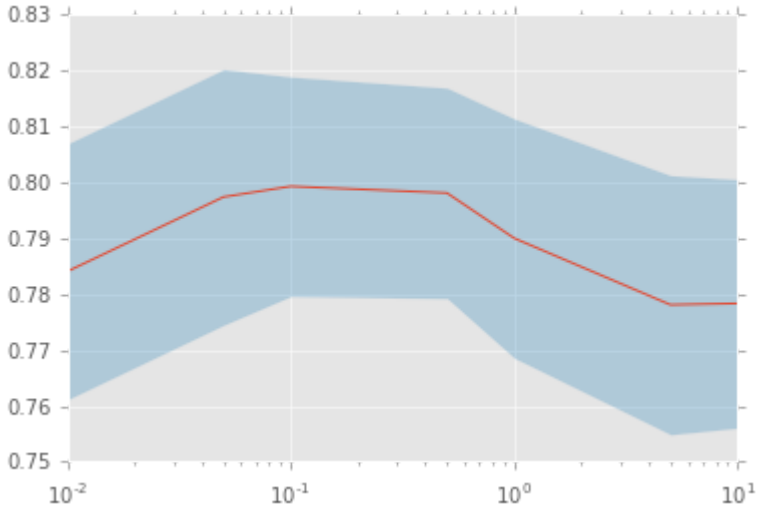
Информация для интересующихся: вообще говоря, не вполне логично оптимизировать на кросс-валидации заданный по умолчанию в классе логистической регрессии функционал accuracy, а измерять на тесте AUC ROC, но это, как и ограничение размера выборки, сделано для ускорения работы процесса кросс-валидации.

In [14]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import roc_auc_score

def plot_scores(optimizer):
    scores = [[item[0]['C'],
               item[1],
               (np.sum((item[2]-item[1])**2)/(item[2].size-1))**0.5] for item in optimizer.grid
    scores = np.array(scores)
    plt.semilogx(scores[:,0], scores[:,1])
    plt.fill_between(scores[:,0], scores[:,1]-scores[:,2],
                                  scores[:,1]+scores[:,2], alpha=0.3)
    plt.show()

def write_answer_1(auc_1, auc_2):
    auc = (auc_1 + auc_2)/2
    with open("preprocessing_lr_answer1.txt", "w") as fout:
        fout.write(str(auc))

param_grid = {'C': [0.01, 0.05, 0.1, 0.5, 1, 5, 10]}
cv = 10

# place your code here
```

In [15]:
```
X_train_cat_oh = pd.DataFrame(X_train_cat_oh)

X_train_cat_oh.index = X_train_real_zeros.index
zero_X = pd.concat([X_train_real_zeros, X_train_cat_oh], axis=1)

X_train_cat_oh.index = X_train_real_mean.index
mean_X = pd.concat([X_train_real_mean, X_train_cat_oh], axis=1)
```

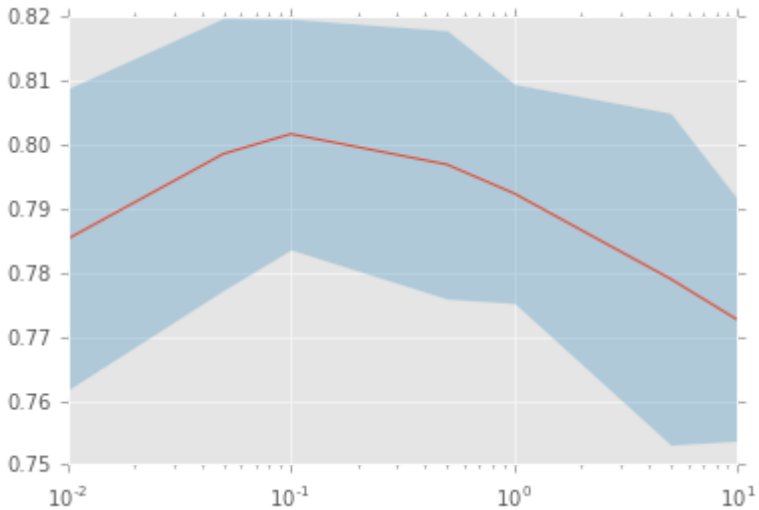
In [16]:
```
estimator_zero = LogisticRegression()
optimizer_zero = GridSearchCV(estimator_zero, param_grid, cv=10, n_jobs=-1)

optimizer_zero.fit(zero_X, y_train)
print 'zeros', optimizer_zero.best_score_
plot_scores(optimizer_zero)
```

```
zeros 0.799285714286
```



In [17]:
```
estimator_mean = LogisticRegression()
optimizer_mean = GridSearchCV(estimator_mean, param_grid, cv=10, n_jobs=-1)

optimizer_mean.fit(mean_X, y_train)
print 'means', optimizer_mean.best_score_
plot_scores(optimizer_mean)
```

```
means 0.801666666667
```



In [18]:
```
score_zero = roc_auc_score(y_test, optimizer_zero.predict_proba(pd.concat([X_test_real_zeros, X
```

In [19]:
```
score_zero
```

Out[19]:
```
0.88681261298940406
```

In [20]:
```
score_mean = roc_auc_score(y_test,optimizer_mean.predict_proba(pd.concat([X_test_real_mean, X_t
```

In [21]:
```
score_mean
```

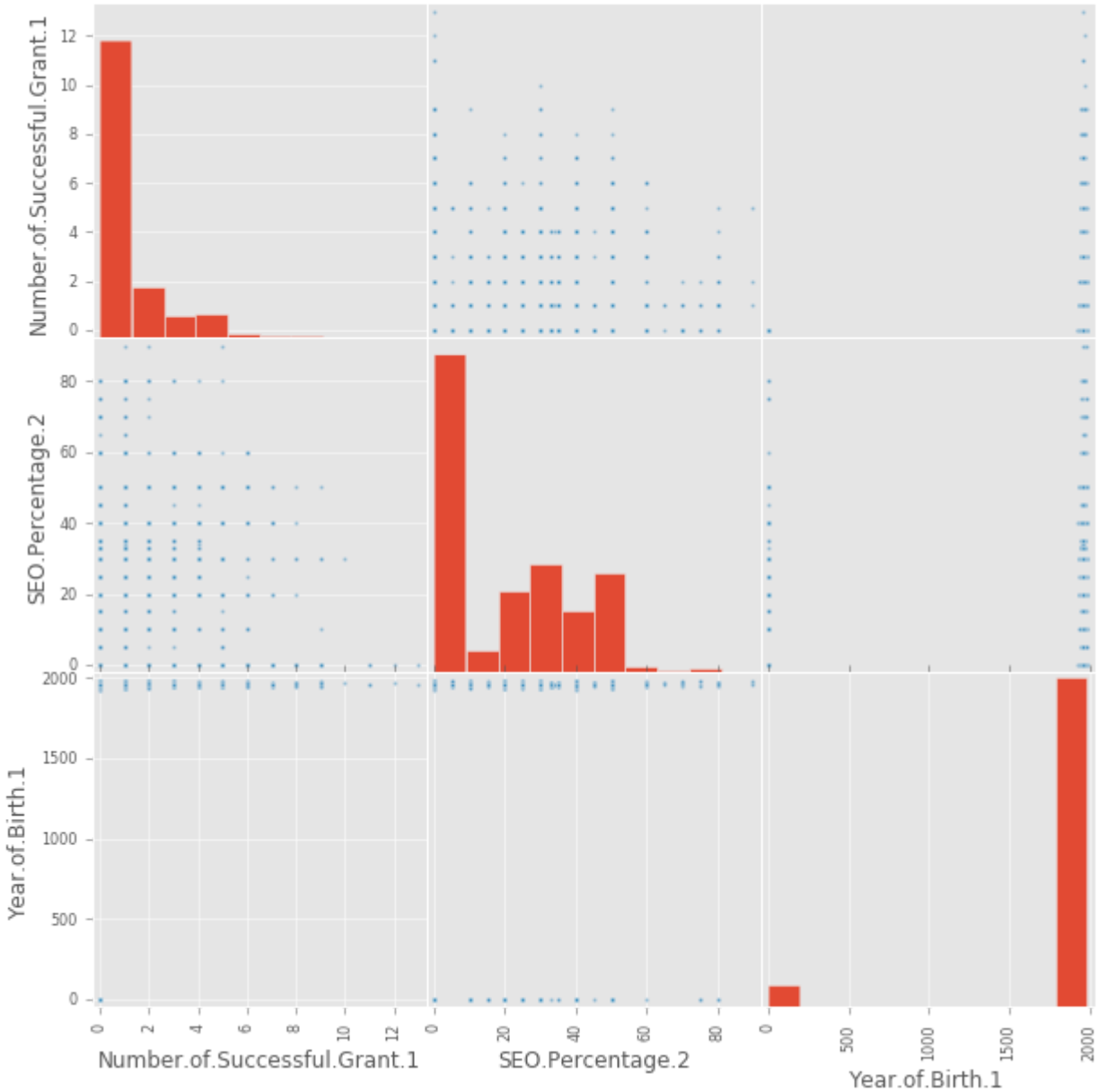
Out[21]:
```
0.88770972256422387
```

In [22]:
```
write_answer_1(score_zero, score_mean)
```

Масштабирование вещественных признаков.

Попробуем как-то улучшить качество классификации. Для этого посмотрим на сами данные:

In [23]:
```
from pandas.tools.plotting import scatter_matrix

data_numeric = pd.DataFrame(X_train_real_zeros, columns=numeric_cols)
list_cols = ['Number.of.Successful.Grant.1', 'SEO.Percentage.2', 'Year.of.Birth.1']
scatter_matrix(data_numeric[list_cols], alpha=0.5, figsize=(10, 10))
plt.show()
```



Как видно из графиков, разные признаки очень сильно отличаются друг от друга по модулю значений (обратите внимание на диапазоны значений осей x и y). В случае обычной регрессии это никак не влияет на качество обучаемой модели, т.к. у меньших по модулю признаков будут большие веса, но при использовании регуляризации, которая штрафует модель за большие веса, регрессия, как правило, начинает работать хуже.

В таких случаях всегда рекомендуется делать стандартизацию (масштабирование) признаков, для того чтобы они меньше отличались друг друга по модулю, но при этом не нарушались никакие другие свойства признакового пространства. При этом даже если итоговое качество модели на тесте уменьшается, это повышает её интерпретабельность, потому что новые веса имеют смысл "значимости" данного признака для итоговой классификации.

Стандартизация осуществляется посредством вычета из каждого признака среднего значения и нормировки на выборочное стандартное отклонение:

$$x_{id}^{scaled} = \frac{x_{id} - \mu_d}{\sigma_d}, \quad \mu_d = \frac{1}{N}\sum_{i=1}^{l} x_{id}, \quad \sigma_d = \sqrt{\frac{1}{N-1}\sum_{i=1}^{l}(x_{id} - \mu_d)^2}$$

Задание 1.5. Масштабирование вещественных признаков.

1. По аналогии с вызовом one-hot encoder примените масштабирование вещественных признаков для обучающих и тестовых выборок X_train_real_zeros и X_test_real_zeros, используя класс

   `StandardScaler`

   и методы

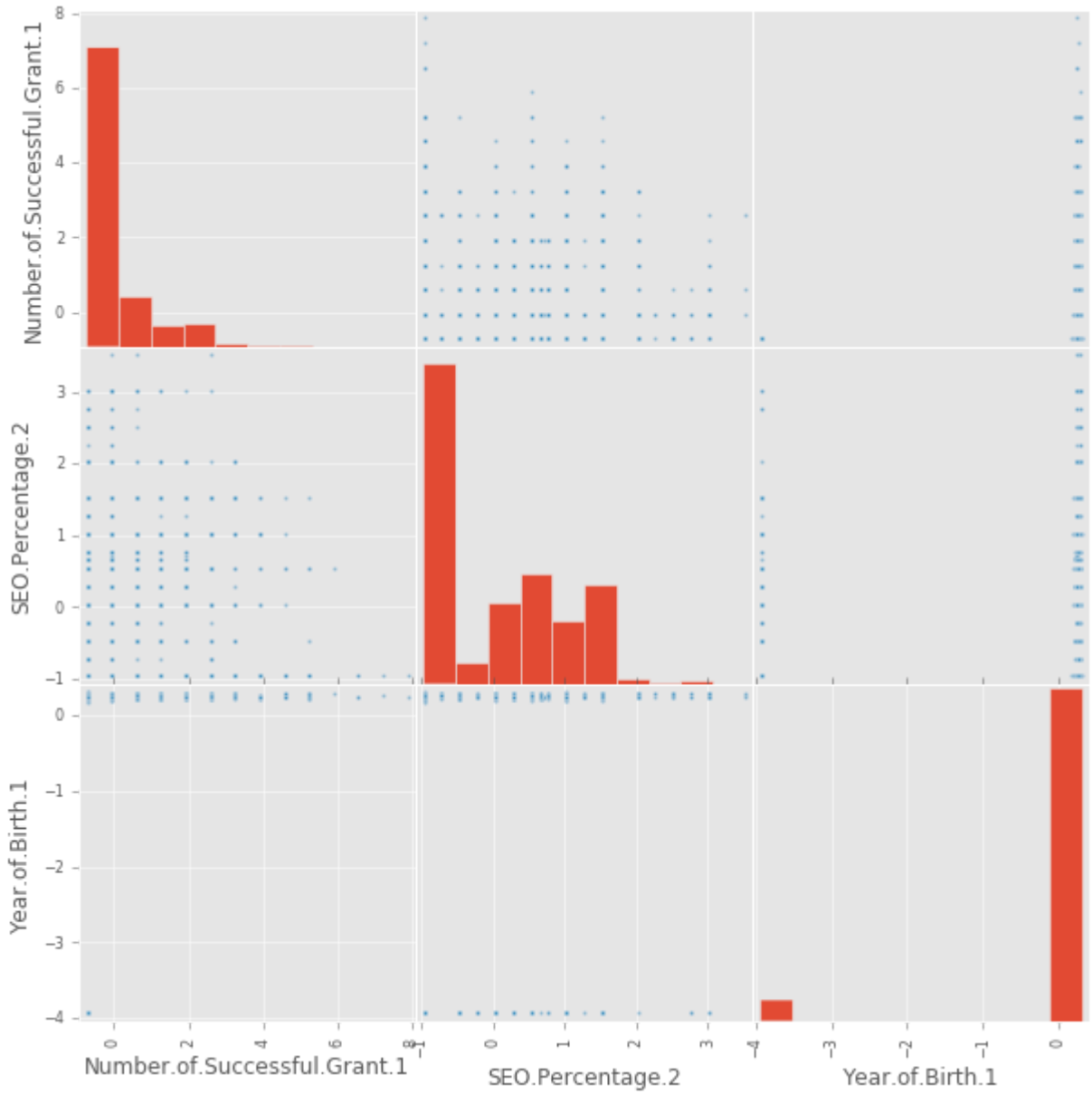
   ```
   StandardScaler.fit_transform(...)
   StandardScaler.transform(...)
   ```
2. Сохраните ответ в переменные X_train_real_scaled и X_test_real_scaled соответственно

In [24]:
```
from sklearn.preprocessing import StandardScaler

# place your code here
scaler = StandardScaler()
X_train_real_scaled = scaler.fit_transform(X_train_real_zeros)
X_test_real_scaled = scaler.transform(X_test_real_zeros)
```

Сравнение признаковых пространств.

Построим такие же графики для преобразованных данных:

In [25]:
```
data_numeric_scaled = pd.DataFrame(X_train_real_scaled, columns=numeric_cols)
list_cols = ['Number.of.Successful.Grant.1', 'SEO.Percentage.2', 'Year.of.Birth.1']
scatter_matrix(data_numeric_scaled[list_cols], alpha=0.5, figsize=(10, 10))
plt.show()
```



Как видно из графиков, мы не поменяли свойства признакового пространства: гистограммы распределений значений признаков, как и их scatter-plots, выглядят так же, как и до нормировки, но при этом все значения теперь находятся примерно в одном диапазоне, тем самым повышая интерпретабельность результатов, а также лучше сочетаясь с идеологией регуляризации.

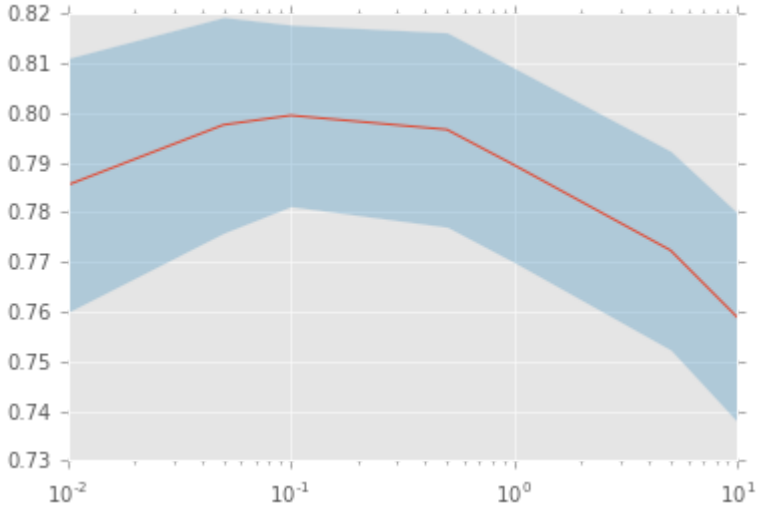
Задание 2. Сравнение качества классификации до и после масштабирования вещественных признаков.

1. Обучите ещё раз регрессию и гиперпараметры на новых признаках, объединив их с закодированными категориальными.
2. Проверьте, был ли найден оптимум accuracy по гиперпараметрам во время кроссвалидации.
3. Получите значение ROC AUC на тестовой выборке, сравните с лучшим результатом, полученными ранее.
4. Запишите полученный ответ в файл при помощи функции write_answer_2.

In [26]:
```
def write_answer_2(auc):
    with open("preprocessing_lr_answer2.txt", "w") as fout:
        fout.write(str(auc))

# place your code here
```

In [27]:
```
X_train_scaled = np.hstack((X_train_real_scaled, X_train_cat_oh))
X_test_scaled = np.hstack((X_test_real_scaled, X_test_cat_oh))
```

In [28]:
```
estimator = LogisticRegression()
optimizer = GridSearchCV(estimator, param_grid, cv=10)
optimizer.fit(X_train_scaled, y_train)
plot_scores(optimizer)
```



In [29]:
```
print 'Zeros', optimizer.best_score_
```

```
Zeros 0.799523809524
```

In [30]:
```
score = roc_auc_score(np.array(y_test), optimizer.predict_proba(X_test_scaled)[:, 1])
```

In [31]:
```
score
```

Out[31]:
```
0.88708483934314253
```

In [32]:
```
write_answer_2(score)
```

In [ ]: