Logistic Regression Variable Selection

Previously we have been building models manually either by having specific variables in mind, or by making targeted comparisons between models with different variables. In this section we begin to explore more automated methods for modeling. The key concept is to embed a a model in a larger class of potential models and tune the models within this class. This tuning process is called **learning** the model, and starts us on the road to machine learning.

A big issue in model selection is the temptation to fit bigger and bigger models in order to improve the fit to the training data. This tendancy is called **overfitting.** By overfitting the data at hand, we risk losing the ability to generalize the results to future data or larger populations, because the model is too fine tuned to the data at hand.

Learning methods are designed to counteract the tendancy to overfit the data. A simple approach introduced in the previous section is to split the data randomly into training and testing subsets of the data. We do all the model building on the training data, and then assess the model using the test data.

The idea is that simpler models might be biased due to some missing variables or transformations, so $E[\hat{Y}] \neq \mu$, but if the bias is not too large compared to the variance reduction they provide, the mean square error can be improved over larger, less biased models with larger variance. If we go too far in this direction, however, the bias will overtake the variance. So we expect there will be some optimal model between the two extremes.

A key modeling aim is to find an effective compromise between bias reduction and variance reduction, for example, by searching for models with small **mean square error** for prediction, such a compromise might be found.

Fundamental bias-variance decomposition for model prediction $\hat{Y}$:

$$MSE(\hat{Y}) = E[(\hat{Y} - \mu)^2] = E[(\hat{Y} - E(\hat{Y}))^2] + [E(\hat{Y}) - \mu]^2$$

$$= Var(\hat{Y}) + Bias^2(\hat{Y}).$$

This section explores several methodologies useful in model selection, aimed at addressing the overfit/underfit challenge:

- **Log-Likelihood-Ratio Tests** for comparing nested logistic regression models; analogous to F-tests in ANOVA
- **Information criteria such as AIC and BIC** that trade off model fit with model complexity
- **Train/Test data splitting** to evaluate model based classifiers for sensitivy, specificity and accuracy

Python libraries and functions:

```
statsmodels.api
statsmodels.formula.api
    logit
scipy.stats
    bernoulli
    chi2
    norm
sklearn.model_selection
    train_test_split
sklearn.metrics
    accuracy_score
    confusion_matrix
    roc_curve
    roc_auc_score
```

Comparing two logit models: log-likelihood ratio test

Recall that in linear regression modeling it can be useful to test between two models using an analysis of variance F test, which compares the residual sums of squares for two, nested models. It allows us to test multiple parameters within one hypothesis test.

In logistic regression modeling, the F test is no longer applicable. However, the same general testing idea is possible by comparing log-likelihoods between two nested models. The change in log-likelihood is used as a large sample chi-square test of the null hypothessis that the simpler model is adequate.

In binary response models such as logistic regression the **likelihood function (LF)** is the joint probability mass function of the responses viewed as a function of the parameters. For a logit model with independent Bernoulli responses, the likelihood function has the form:

$$LF(\beta_0, \beta_1, \ldots, \beta_p) = \prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1-y_i}$$

where

$$\log\left(\frac{p_i}{1 - p_i}\right) = \beta_0 + \beta_1 X_{i1} + \cdots \beta_p X_{ip}, \quad \text{for} \quad i = 1, 2, \ldots, n.$$

The logarithmic tranformation converts the product to a sum of log values, the log-likelihood function (LLF):

$$LLF(\beta_0, \beta_1, \ldots, \beta_p) = \sum_{i=1}^{n} \{y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\}.$$

The result reported in the model summary is the optimized value computed by **maximum likelihood estimation**:

$$llf = LLF(\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p) = \sum_{i=1}^{n} \{y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)\}$$

General result for comparing nested models

In order to test between two logit models, Model 0 and Model 1, where Model 0 is a special case of Model 1 obtained by setting some regression coefficients equal to zero. Consider one is a special case of the other we can compare their log-likelihood ratios. Consider testing:

$$H_0: \quad \text{Model 0 is correct,}$$

$$H_A: \quad \text{Model 0 is incorrect because at least one missing coefficient from Model 1 is not zero.}$$

A general result from large sample theory is if H_0 is true, then twice the difference in negative log-likelihoods

$$llr = -2\,(llf_0 - llf_1)$$

has an approximate Chi-square distribution with degrees of freedom equal to the difference in the numbers of parameters for the two models. Like the central limit theorem, this approximation works better for larger sample size n.

Applying this test in our example lets us evaluate multiple coefficients at the same time to determine whether we can reduce to the simpler model. Here's how it works.

Example: comparing two logit models for Pew Research Survey data

In an earlier section we considered two models for predicting a favorable opinion of border wall construction in the Pew Research Survey of February 2017. Let's load the data and the two models and first see how we can test between the two models. The idea is analogous to the ANOVA method for comparing two linear regression models.

Preprocessing and data validation

In [1]:
```
import numpy as np
import pandas as pd
import zipfile as zp
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

In [2]:
```
zf = zp.ZipFile('../data/Feb17-public.zip')
missing_values = ["NaN", "nan", "Don't know/Refused (VOL.)"]
df = pd.read_csv(zf.open('Feb17public.csv'),
                 na_values=missing_values)[['age', 'sex', 'q52', 'party'
]]
```

In [3]:
```
# reduce q52 responses to two categories
# and create binary reponse variable
df['q52'][df['q52']!='Favor'] = 'Not_favor'
df['y'] = df['q52'].map({'Not_favor':0,'Favor':1})
# use cleaned data without records that have missing values
dfclean = df.dropna()
```

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

Out[4]:

	age	sex	q52	party	y
0	80.0	Female	Not_favor	Independent	0
1	70.0	Female	Not_favor	Democrat	0
2	69.0	Female	Not_favor	Independent	0
3	50.0	Male	Favor	Republican	1
4	70.0	Female	Not_favor	Democrat	0

In [5]:
```
dfclean['party'].value_counts()
```

Out[5]:
```
Democrat                  527
Independent               525
Republican                367
No preference (VOL.)       41
Other party (VOL.)          5
Name: party, dtype: int64
```

In [6]:
```
dfclean['sex'].value_counts()
```

Out[6]:
```
Male      760
Female    705
Name: sex, dtype: int64
```

```
In [7]: dfclean.describe()
```

Out[7]:

	age	y
count	1465.000000	1465.000000
mean	50.522867	0.341297
std	17.843611	0.474307
min	18.000000	0.000000
25%	35.000000	0.000000
50%	52.000000	0.000000
75%	65.000000	1.000000
max	96.000000	1.000000

```
In [8]: dfclean.groupby('party').mean()
```

Out[8]:

	age	y
party		
Democrat	50.499051	0.077799
Independent	46.807619	0.306667
No preference (VOL.)	43.146341	0.317073
Other party (VOL.)	44.600000	0.600000
Republican	56.776567	0.768392

We can see that the proportion of 'favor' responses varies quite a bit between party affiliations, by looking at the mean values for 'y'. In each subgroup, the sample mean of y equals the proportion who favored building the wall.

Is it statistically signicant? We can test this using a log-likelihood-ratio test.

Full model and reduced model for log-likelihood-ratio test

Recall that 'party' is a categorical variable with 5 categories. If we wish to test the null hypothesis of no party effects, we need a 4 degree of freedom test. For this we can use the log-likelihood-ratio test.

First we fit the null and full model:

In [9]:
```
model0 = smf.logit('y ~ age + sex', data=dfclean).fit()
model1 = smf.logit('y ~ party + age + sex', data=dfclean).fit()
```

```
Optimization terminated successfully.
         Current function value: 0.619057
         Iterations 5
Optimization terminated successfully.
         Current function value: 0.466129
         Iterations 6
```

We don't need to display the summaries to perform the test, but it is informative to review the model summaries to understand the variables. The maximized log-likelihood is shown in the model summary as 'llf'.

In [10]:
```
model0.summary()
```

Out[10]:

Logit Regression Results

Dep. Variable:	y	**No. Observations:**	1465
Model:	Logit	**Df Residuals:**	1462
Method:	MLE	**Df Model:**	2
Date:	Mon, 20 Apr 2020	**Pseudo R-squ.:**	0.03557
Time:	13:44:11	**Log-Likelihood:**	-906.92
converged:	True	**LL-Null:**	-940.37
Covariance Type:	nonrobust	**LLR p-value:**	2.960e-15

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
Intercept	-2.0818	0.196	-10.637	0.000	-2.465	-1.698
sex[T.Male]	0.5415	0.114	4.750	0.000	0.318	0.765
age	0.0220	0.003	6.770	0.000	0.016	0.028

In [11]:
```
model1.summary()
```
Out[11]:

Logit Regression Results

Dep. Variable:	y	**No. Observations:**	1465
Model:	Logit	**Df Residuals:**	1458
Method:	MLE	**Df Model:**	6
Date:	Mon, 20 Apr 2020	**Pseudo R-squ.:**	0.2738
Time:	13:44:11	**Log-Likelihood:**	-682.88
converged:	True	**LL-Null:**	-940.37
Covariance Type:	nonrobust	**LLR p-value:**	4.971e-108

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-3.5261	0.281	-12.536	0.000	-4.077	-2.975
party[T.Independent]	1.6843	0.191	8.796	0.000	1.309	2.060
party[T.No preference (VOL.)]	1.8226	0.379	4.807	0.000	1.079	2.566
party[T.Other party (VOL.)]	2.8930	0.938	3.083	0.002	1.054	4.732
party[T.Republican]	3.5862	0.206	17.435	0.000	3.183	3.989
sex[T.Male]	0.3721	0.137	2.712	0.007	0.103	0.641
age	0.0168	0.004	4.305	0.000	0.009	0.024

Here's how we can extract the log-likelihoods for the two models:

In [12]:
```
model0.llf, model1.llf
```
Out[12]:
```
(-906.9182356126391, -682.8795444475213)
```

In [13]:
```
model0.df_model, model1.df_model
```
Out[13]:
```
(2.0, 6.0)
```

Compare log-likelihoods and perform likelihood ratio statistic.

Just be careful to get the multiplier (-2) right so the chi-sqaure approximation works correctly.

In [14]:

```python
# Extract log-likelihood function values
# and model degrees of freedom from each model
llf0, df0 = model0.llf, model0.df_model
llf1, df1 = model1.llf, model1.df_model
# take differences
llr, dfdiff = -2*(llf0 - llf1), df1 - df0
# display results
pd.DataFrame({'-2*llf': [-2*llf0, -2*llf1, llr],
              'df_model': [df0, df1, dfdiff]},
             index=['model0','model1', 'diff'])
```

Out[14]:

	-2*llf	df_model
model0	1813.836471	2.0
model1	1365.759089	6.0
diff	448.077382	4.0

In [15]:

```python
# import chisquare function and compute p-value
from scipy.stats import chi2
1 - chi2.cdf(llr, df=dfdiff)
```

Out[15]: 0.0

Summarize the test with calculated p-value using chi-square distribution

In [16]:

```python
# summarize test results
print('-2*llr:', round(llr, 2), \
      ' df:', dfdiff, ' p-value:', \
      1 - chi2.cdf(llr, df=dfdiff))
```

```
-2*llr: 448.08  df: 4.0  p-value: 0.0
```

Conclusion: We definitely reject the null hypothesis and favor Model 1 over Model 0. Party affiliation is a significant factor associated with the response to question 52 in the survey.

Model selection criteria: AIC and BIC

AIC and BIC are criteria for evaluating a model that combine the likelihood asessment of fit with a penalty for complex models. Historically they were derived from different perspectives. The Akaike Information Criterion (AIC), has the form

$$\text{AIC} = -2 * llf + 2 * \frac{p}{n},$$

where p is the same as the model degrees of freedom. Small values are considered better than large values, so minimizing AIC favors larger likelihoods and simpler models, while trying to balance these two goals.

The Bayesian Information Criterion (BIC) is related but uses different relative weighting of likelihood and complexity:

$$\text{BIC} = -2 * llf + p * \log(n).$$

Again, models with smaller values are better than models with larger values. Both methods enforce favoring simpler models among those with simimlar fit overall, and they help prevent overfitting the model because of the complexity penalty.

In the current implementation of the statsmodels logit api, both of these criteria are available from the model fitting results. Here's a summary for our two models of the Pew survey data for predicting favorable or unfavorable opinions of the border wall:

In [17]:
```
model1.aic, model0.aic, model1.bic, model0.bic
```

Out[17]:
```
(1379.7590888950426,
 1819.8364712252783,
 1416.7863625452007,
 1835.7053027896318)
```

In [18]:
```
pd.DataFrame({'-2*llf': [-2*llf0, -2*llf1],
              'df_model': [df0, df1],
              'AIC': [model0.aic, model1.aic],
              'BIC': [model0.bic, model1.bic]},
             index=[0,1])
```

Out[18]:

	-2*llf	df_model	AIC	BIC
0	1813.836471	2.0	1819.836471	1835.705303
1	1365.759089	6.0	1379.759089	1416.786363

Conclusion: Both AIC and BIC favor Model 1. This suggests that Model 0 is too simple, so the bias due to omitted variables is too large for this model compared to Model 1.

Use cases: AIC and/or BIC are often used to guide variable selection when multiple exogensous variables are considered for inclusion in the model. This enables us to compare a whole series of models and try to find a reasonable tradeoff between bias and variance, i.e., goodness of fit and model complexity. BIC tends to favor simplicity more heavily than does AIC due to its heavier penalty for large p.

Evaluation of predictive accuracy: Although model selection criteria like AIC and BIC can help avoid overfitting and underfitting the data, they do no provide us with assessment of classification performance. In order to evaluate the model selected by these criteria or related strategies, it is still necessary to use some version of the train/test method, where the training data are used for the model building process, and the test data are reserved for predictive evaluation only.

Simulation example with many variables

Generate the features matrix and response data from a logit model

For illustration in an example with many explanatory variables we generate binary response data with 20 explanatory variables. First we set up the coefficient vector for the simulation model.

In [19]:
```
from scipy.stats import norm, bernoulli
from sklearn.model_selection import train_test_split
```

In [20]:
```
## make a coefficient vector for logit model
b0 = -1  # intercept
bvec = np.repeat([2,-1.5,0.5], [5, 5, 10])  # feature coefficients
bvec
```

Out[20]:
```
array([ 2. ,  2. ,  2. ,  2. ,  2. , -1.5, -1.5, -1.5, -1.5, -1.5,  0.
5,
        0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5])
```

Next we generate a random features matrix, using numpy matrix operations to form the matrix.

In [21]:
```
# generate a features matrix with n observations
# and columns matching the coefficent vector
n = 200
nX = bvec.size
X = norm.rvs(size=n*nX, random_state=1).reshape((n, nX))
X.shape
```

Out[21]:
```
(200, 20)
```

In [22]:
```
X
```

Out[22]:
```
array([[ 1.62434536, -0.61175641, -0.52817175, ..., -0.87785842,
         0.04221375,  0.58281521],
       [-1.10061918,  1.14472371,  0.90159072, ...,  0.2344157 ,
         1.65980218,  0.74204416],
       [-0.19183555, -0.88762896, -0.74715829, ...,  0.93110208,
         0.28558733,  0.88514116],
       ...,
       [ 1.99151525,  1.29962918, -0.59207261, ..., -0.45165125,
        -0.52973059,  0.63291748],
       [ 0.87499606, -1.04936913, -0.60735181, ..., -0.10561872,
        -0.86173477,  0.47313567],
       [-0.13888137,  2.65213968, -0.656247  , ..., -0.84391327,
         0.62834172,  0.53721449]])
```

Use numpy matrix multiplication to form the log-odds model, and exponentiate to get the vector of n odds for the responses.

In [23]:
```
# compute the odds of 1 for n observations
# use numpy matrix multiplication to make this easier
odds = np.exp(b0 + np.matmul(X, bvec))
odds.shape
```

Out[23]:
```
(200,)
```

Convert the odds vector to the population probability vector for the n 0/1 responses. Then use the bernoulli.rvs function to generate the responses from the model.

In [24]:
```
# compute simulated Bernoulli responses
y = bernoulli.rvs(p=odds/(1+odds), size=n, random_state=12347)
y.shape
```

Out[24]:
```
(200,)
```

In [25]:
```
y[0:20]
```

Out[25]:
```
array([0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1])
```

To set up for formula based modeling, we assign names to the columns of X.

In [26]:
```
# Give the features names and load X into a data frame
Xnames = []
for i in range(nX):
    list.append(Xnames, 'X'+str(i+1))
df = pd.DataFrame(X, columns=Xnames)
```

In [27]:
```
# Add y to the data frame
df['y'] = y
display(df.shape, df.iloc[0:5,:7], \
        df.iloc[0:5, 7:14], df.iloc[0:5, 15:])
```

```
(200, 21)
```

	X1	X2	X3	X4	X5	X6	X7
0	1.624345	-0.611756	-0.528172	-1.072969	0.865408	-2.301539	1.744812
1	-1.100619	1.144724	0.901591	0.502494	0.900856	-0.683728	-0.122890
2	-0.191836	-0.887629	-0.747158	1.692455	0.050808	-0.636996	0.190915
3	-0.754398	1.252868	0.512930	-0.298093	0.488518	-0.075572	1.131629
4	-0.222328	-0.200758	0.186561	0.410052	0.198300	0.119009	-0.670662

	X8	X9	X10	X11	X12	X13	X14
0	-0.761207	0.319039	-0.249370	1.462108	-2.060141	-0.322417	-0.384054
1	-0.935769	-0.267888	0.530355	-0.691661	-0.396754	-0.687173	-0.845206
2	2.100255	0.120159	0.617203	0.300170	-0.352250	-1.142518	-0.349343
3	1.519817	2.185575	-1.396496	-1.444114	-0.504466	0.160037	0.876169
4	0.377564	0.121821	1.129484	1.198918	0.185156	-0.375285	-0.638730

	X16	X17	X18	X19	X20	y
0	-1.099891	-0.172428	-0.877858	0.042214	0.582815	0
1	-0.012665	-1.117310	0.234416	1.659802	0.742044	1
2	0.586623	0.838983	0.931102	0.285587	0.885141	0
3	-2.022201	-0.306204	0.827975	0.230095	0.762011	0
4	0.077340	-0.343854	0.043597	-0.620001	0.698032	0

Split the data into training data and test data

In [28]:
```
# split the data frame into training data (traindf)
# and testing data (testdf)
df_train, df_test = train_test_split(
    df, test_size=0.20, random_state=42)
```

In [29]:
```
df_train.shape, df_test.shape
```

Out[29]:
```
((160, 21), (40, 21))
```

Model the training data: are 20 variables necessary?

In [30]:
```
mod0 = smf.logit(
    'y ~ X1+X2+X3+X4+X5+X6+X7+X8+X9+X10\
    +X11+X12+X13+X14+X15+X16+X17+X18+X19+X20',
    data=df_train).fit()
```

```
Optimization terminated successfully.
         Current function value: 0.176600
         Iterations 10
```

In [31]:
```
# model information
mod0.summary().tables[0]
```

Out[31]:

Logit Regression Results

Dep. Variable:	y	**No. Observations:**	160
Model:	Logit	**Df Residuals:**	139
Method:	MLE	**Df Model:**	20
Date:	Mon, 20 Apr 2020	**Pseudo R-squ.:**	0.7376
Time:	13:44:12	**Log-Likelihood:**	-28.256
converged:	True	**LL-Null:**	-107.68
Covariance Type:	nonrobust	**LLR p-value:**	1.251e-23

In [32]:
```
# model coefficient summary table
mod0.summary().tables[1]
```

Out[32]:

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-1.3640	0.504	-2.709	0.007	-2.351	-0.377
X1	2.4668	0.700	3.524	0.000	1.095	3.839
X2	2.5231	0.752	3.355	0.001	1.049	3.997
X3	2.6866	0.737	3.645	0.000	1.242	4.131
X4	2.1278	0.767	2.773	0.006	0.624	3.632
X5	2.1866	0.618	3.541	0.000	0.976	3.397
X6	-1.3311	0.503	-2.645	0.008	-2.317	-0.345
X7	-2.4016	0.653	-3.677	0.000	-3.682	-1.121
X8	-1.6166	0.576	-2.806	0.005	-2.746	-0.487
X9	-2.0160	0.710	-2.838	0.005	-3.409	-0.624
X10	-2.2436	0.721	-3.112	0.002	-3.657	-0.831
X11	1.4669	0.551	2.660	0.008	0.386	2.548
X12	0.7205	0.432	1.670	0.095	-0.125	1.566
X13	0.8124	0.451	1.802	0.072	-0.071	1.696
X14	0.1165	0.412	0.283	0.777	-0.690	0.923
X15	0.2603	0.459	0.567	0.571	-0.640	1.160
X16	0.6365	0.489	1.302	0.193	-0.321	1.594
X17	0.3760	0.392	0.960	0.337	-0.392	1.144
X18	-0.1898	0.545	-0.348	0.728	-1.258	0.878
X19	1.0728	0.477	2.248	0.025	0.137	2.008
X20	0.8356	0.489	1.710	0.087	-0.122	1.793

Here are AIC and BIC for the model:

In [33]:
```
(mod0.aic, mod0.bic)
```

Out[33]:
```
(98.51189376685147, 163.09054388676185)
```

Let's compare a simpler model. There are many possible models (2^{20}), so how can we process them? An old idea is to use the coefficient tests to help filter variables.

In [34]:
```
mod0.pvalues.sort_values()
```

Out[34]:
```
X7           0.000236
X3           0.000267
X5           0.000399
X1           0.000425
X2           0.000794
X10          0.001857
X9           0.004545
X8           0.005020
X4           0.005551
Intercept    0.006752
X11          0.007813
X6           0.008164
X19          0.024573
X13          0.071623
X20          0.087216
X12          0.095010
X16          0.192819
X17          0.337042
X15          0.570756
X18          0.727601
X14          0.777220
dtype: float64
```

In [35]:
```
mod0.pvalues[mod0.pvalues < 0.05]
```

Out[35]:
```
Intercept    0.006752
X1           0.000425
X2           0.000794
X3           0.000267
X4           0.005551
X5           0.000399
X6           0.008164
X7           0.000236
X8           0.005020
X9           0.004545
X10          0.001857
X11          0.007813
X19          0.024573
dtype: float64
```

Let's compare the model that only keeps these "significant" variables.

In [36]:
```
mod1 = smf.logit(
    'y ~ X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X19',
    data=df_train).fit()
(mod1.aic, mod1.bic)
```

```
Optimization terminated successfully.
         Current function value: 0.227307
         Iterations 9
```

Out[36]:
```
(98.7382939405669, 138.71555353860666)
```

BIC is reduced. AIC is about the same. Let's try to go further.

In [37]:
```
mod1.pvalues[mod1.pvalues < 0.05]
```

Out[37]:
```
Intercept    0.010066
X1           0.000045
X2           0.000043
X3           0.000013
X4           0.000229
X5           0.000192
X6           0.008512
X7           0.000085
X8           0.000584
X9           0.000647
X10          0.000220
X11          0.002939
X19          0.023514
dtype: float64
```

In [38]:
```
# least significant variable in mod1
mod1.pvalues[mod1.pvalues==max(mod1.pvalues)]
```

Out[38]:
```
X19    0.023514
dtype: float64
```

Try dropping this least significant variable to see what happens.

In [39]:
```
mod2 = smf.logit('y ~ X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11',
                 data=df_train).fit()
(mod2.aic, mod2.bic)
```

```
Optimization terminated successfully.
         Current function value: 0.245458
         Iterations 9
```

Out[39]:
```
(102.5466613953034, 139.44874717810933)
```

AIC and BIC increase a bit. Try dropping one more...

In [40]:
```
mod2.pvalues[mod2.pvalues==max(mod2.pvalues)]
```

Out[40]:
```
Intercept    0.013388
dtype: float64
```

In [41]:
```
mod2.pvalues.sort_values()
```

Out[41]:
```
X3           0.000015
X2           0.000018
X1           0.000032
X7           0.000081
X5           0.000210
X10          0.000236
X4           0.000363
X8           0.000818
X9           0.001048
X11          0.005432
X6           0.008977
Intercept    0.013388
dtype: float64
```

What happens if we drop X6?

In [42]:
```
mod3 = smf.logit('y ~ X1+X2+X3+X4+X5+X7+X8+X9+X10',
                 data=df_train).fit()
(mod3.aic, mod3.bic)
```

```
Optimization terminated successfully.
         Current function value: 0.297416
         Iterations 8
```

Out[42]:
```
(115.17324407941844, 145.9249822317567)
```

Even more increase. Looks like we can't reduce the model beyond mod1, based on these criteria.

In [43]:
```
# Summarize results
pd.DataFrame({'aic': [mod0.aic, mod1.aic, mod2.aic, mod3.aic],
             'bic': [mod0.bic, mod1.bic, mod2.bic, mod3.bic] },
             index=[0,1,2,3])
```

Out[43]:

	aic	bic
0	98.511894	163.090544
1	98.738294	138.715554
2	102.546661	139.448747
3	115.173244	145.924982

According to BIC, model 1 is the best. According to AIC it's very close between mod0 and mod1. Here's the model summary for mod1:

In [44]:
```
mod1.summary()
```

Out[44]:

Logit Regression Results

Dep. Variable:	y	**No. Observations:**	160
Model:	Logit	**Df Residuals:**	147
Method:	MLE	**Df Model:**	12
Date:	Mon, 20 Apr 2020	**Pseudo R-squ.:**	0.6623
Time:	13:44:12	**Log-Likelihood:**	-36.369
converged:	True	**LL-Null:**	-107.68
Covariance Type:	nonrobust	**LLR p-value:**	1.766e-24

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
Intercept	-0.9374	0.364	-2.574	0.010	-1.651	-0.223
X1	2.1566	0.529	4.080	0.000	1.120	3.193
X2	2.0498	0.501	4.090	0.000	1.068	3.032
X3	1.9472	0.447	4.358	0.000	1.071	2.823
X4	1.7873	0.485	3.685	0.000	0.837	2.738
X5	1.4869	0.399	3.729	0.000	0.705	2.268
X6	-0.9173	0.349	-2.631	0.009	-1.601	-0.234
X7	-1.9646	0.500	-3.930	0.000	-2.944	-0.985
X8	-1.4560	0.423	-3.439	0.001	-2.286	-0.626
X9	-1.7072	0.500	-3.411	0.001	-2.688	-0.726
X10	-1.7636	0.477	-3.695	0.000	-2.699	-0.828
X11	1.1117	0.374	2.974	0.003	0.379	1.844
X19	0.7941	0.351	2.265	0.024	0.107	1.481

Compared to the simulation model that generated the data we see that the best fitted model is missing X8 and includes X14, which we know to have a zero coefficient from the simulation model. This is an example of the effects of sample variation in model building.

Evaluate selected model as a classifier on test data

Let's compute the accuracies of the models as classifiers. We'll use the predictive probability as the classification score use the classification rule:

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ 0, & \text{if } \hat{p} < 0.5 \end{cases}$$

We compute the predictive probabilities for Model 1:

In [45]:
```
phat1 = mod1.predict(exog=df_test)
phat1[0:10]
```

Out[45]:
```
95     0.347583
15     0.050317
30     0.001393
158    0.999849
128    0.016493
115    0.001555
69     0.000541
170    0.000013
174    0.020079
45     0.079501
dtype: float64
```

Classification accuracy

Here's a function to compute the classification accuracy, which is the overall fraction correctly classified.

In [46]:
```
from sklearn.metrics import accuracy_score
```

In [47]:
```
pthresh = 0.5
accuracy_score(y_true=df_test['y'],
               y_pred=1*(phat1 >= pthresh),
               normalize=True)
```

Out[47]:
```
0.925
```

The classification accuracy for Model 1 is estimated to be 92.5%. This is the combination of the true positives rate and the true negatives rate, if we view 1's as positive and 0's as negative. How does this compare to the other models?

In [48]:
```
# bind together all the predictive probabilities into a matrix
phat_matrix = np.array([mod0.predict(exog=df_test),
                        mod1.predict(exog=df_test),
                        mod2.predict(exog=df_test),
                        mod3.predict(exog=df_test)])
phat_matrix.shape
```

Out[48]:
```
(4, 40)
```

In [49]:
```
phat_matrix[2][0:10]  # compare with values above
```

Out[49]:
```
array([3.09116764e-01, 1.02689627e-01, 1.21609512e-03, 9.99795672e-01,
       1.93365814e-02, 6.22526164e-03, 6.48536897e-04, 1.61225639e-05,
       2.16677944e-02, 8.21426220e-02])
```

In [50]:
```
accuracy_list = []
for i in range(0,4):
    accuracy_list.append(
        accuracy_score(y_true=df_test['y'],
                       y_pred=1*(phat_matrix[i] >= pthresh),
                       normalize=True)
    )
```

In [51]:
```
accuracy_list
```

Out[51]:
```
[0.875, 0.925, 0.875, 0.875]
```

In [52]:
```
pd.DataFrame({'aic': [mod0.aic, mod1.aic, mod2.aic, mod3.aic],
             'bic': [mod0.bic, mod1.bic, mod2.bic, mod3.bic],
             'accuracy': accuracy_list},
             index=[0,1,2,3])
```

Out[52]:

	aic	bic	accuracy
0	98.511894	163.090544	0.875
1	98.738294	138.715554	0.925
2	102.546661	139.448747	0.875
3	115.173244	145.924982	0.875

For this test set there is no much difference between these models in tersm of classification accuracy, though the model with smallest AIC had the highest accuracy.

Sensitivity, specificity and accuracy

Accuracy is a blunt measure that depends on the overall fraction of each category as well as the sensitivity and specificity. We can break out the component sensitivity and specificity as illustrated in th previous section.

Here's a function used in the previous section for that purpose, modified to include accuracy, and to return a single row data frame.

In [53]:
```
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score
```

In [54]:
```
def senspec(y, score, thresh, index=0):
    yhat = 1*(score >= thresh)
    tn, fp, fn, tp = confusion_matrix(y_true=y, y_pred=yhat).ravel()
    sens = tp / (fn + tp)
    spec = tn / (fp + tn)
    accuracy = (tn+tp)/(tn+fp+fn+tp)
    return pd.DataFrame({'tn':[tn],
                         'fp':[fp],
                         'fn':[fn],
                         'tp':[tp],
                         'sens':[sens],
                         'spec':[spec],
                         'accuracy':[accuracy]})
```

In [55]:
```
# sensitivy and specificity for Model 0
senspec(df_test['y'], phat_matrix[0], 0.5)
```

Out[55]:

	tn	fp	fn	tp	sens	spec	accuracy
0	22	1	4	13	0.764706	0.956522	0.875

In [56]:
```
# sensitivity and specificity for all four models
perf = senspec(df_test['y'], phat_matrix[0], 0.5)
for i in range(1,4):
    temp = perf.append(senspec(df_test['y'], phat_matrix[i], 0.5),
                       ignore_index=True)
    perf = temp
perf
```

Out[56]:

	tn	fp	fn	tp	sens	spec	accuracy
0	22	1	4	13	0.764706	0.956522	0.875
1	23	0	3	14	0.823529	1.000000	0.925
2	22	1	4	13	0.764706	0.956522	0.875
3	22	1	4	13	0.764706	0.956522	0.875

The accuracy, sensitivity and specificity are all better for Model 1 (minimum BIC model) versus the others.

Remark on the selection of variables

From the simulation model we know that all 20 variables had some nonzero population coefficients, so why are some not significant? And why are they removed by the AIC/BIC criteria?

- First, note that the effect sizes for variables X11-X20 are small compared to the effects of X1-X10. With the sample size of 200, small effects often are not statistically significant due to the large standard errors compared to the estimates. We don't have enough power to detect those small effects.
- Second, the predictive performance of the model can be sometimes be improved by removing seemingly significant variables due to the reduced burden of estimation. Having fewer coefficients to estimate can decrease variance and improve mean square error for prediction as long as we retain enough highly informative variables.
