Unit 15: Machine Learning, Regularized Logistic Regression, and Cross-Validation

Case Studies:

- We will use simulated data to introduce the following concepts:
    - Basics of machine learning
    - Regularized Logistic Regression
    - Cross-Validation

Summary of Concepts:

- **Definitions/Properties**
    - Objective function
    - Regularized Regression
    - Cross-validation
- **Modeling**
    - **How can we choose the "best" explanatory variables to use in a logistic regression model (without having to fit multiple models)?**
        - What is the objective function that we are optimizing in "basic" logistic regression?
        - What are some methods that we can modify the "basic" objective function for logistic regression to "delete" unnecessary explanatory variables?
        - What are the pros and cons of each of these methods?
    - **How can we fully utilize the *whole* dataset when training and testing a model?**

Unit 15: Machine Learning, Regularized Logistic Regression, and Cross-Validation

Topic 1: Introduction to Regularized Logistic Regression

Review of Methods (we already learned) to Select the "Best" Explanatory Variables in a Logistic Regression

What methods have we learned so far that allow for us to select the "best" explanatory variables in a model? How do we define "best" in each of these methods?

What is a common downside of each of these methods?

Main Idea of Regularized Regression

Main difference between regularized regression vs. regression?

Regularized regression methods are designed to improve modeling of high dimensional data by incorporating a penalty function to drive variable selection. The Python Scikit-Learn library includes regularized regression and regularized logistic regression as standard model optimization methods. This section will provide a brief overview of penalized logistic regression and demonstrate how the methods work on simulated data.

Topic 2: Introduction to Cross-Validation

Review of Methods (we already learned) to Evaluate Classification Accuracy of "New Data" with Regression Models

What methods have we learned so far that allow for us to evaluate the classification accuracy of "new data" in logistic regression models?

What is a downside of this method?

Main Idea of Cross-Validation

In order to evaluate the classification accuracy of regularized logistic regression we can use the Train/Test method to avoid optimistic bias. One limitation is that it uses only part of the data for both model building and for assessment.

An extension of simple Train/Test is **cross-validation.** This method uses all of the data by repeated application of train/test splitting. The results are then averaged to improve the accuracy. The approach also gives us a simple way to estimate the uncertainty of the accuracy estimate.

Common Types of Cross-Validation

Type 1: Leave-One-Out Cross-Validation

The oldest such method is **leave-one-out cross-validation**, where each of the n observations in a sample is omitted in turn. The estimate is recomputed for each of the n subsets of size $n-1$. The resulting n predictions are then used to estimate predictive accuracy.

Type 2: k-Fold Cross-Validation

Modern methods use **k-fold cross-validation**, where the data are split into k subsets. $k-1$ subsets are used for training, and the left out subset is used for testing. This is repeated at least k times, leaving out each of the k subsets in turn.

The random selection of the k subsets can itself be repeated to generate multiple k fold splits, thus avoiding over dependence on one particular random split.

To use these methods efficiently in Python, we first discuss the structure of models in the machine learning library Scikit-Learn. Then we illustrate some of the cross validation capabilities in the library.

Topic 3: Introduction to Machine Learning

3a: Model building with Scikit-Learn Package (as opposed to statsmodels package)

Regularized Logistic Regression is a type of machine learning algorithm. In order to build a *regularized* logistic regression model, we need to use a different Python package.

Libraries and modules

```
pandas, numpy
matplotlib.pyplot
seaborn
    heatmap
scipy.stats
    norm, bernoulli
sklearn.linear_model
    LogisticRegression
sklearn.model_selection
    train_test_split
sklearn.metrics
    confusion_matrix
    roc_curve, roc_auc_score
    cross_val_score
```

What is the difference in the type of data that statsmodel package uses vs. the type of data that scikit-learn package uses?

Unlike the statsmodels API, which is formula based and operates on pandas data frames, the machine learning functions in the Scikit-Learn expect data to be encoded as numerical matrices.

Machine Learning Definitions: Two Main Matrices

Features Matrix

The **features matrix** X is an array with n rows of observations and p columns of features (explanatory variables);

We can think of the features matrix as a data frame containing only explanatory variables. Sci-kit learn modules generally expect X to contain only numerical columns. Therefore it provides preprocessing functions to

- Convert categorical and boolean data into numerically coded columns (e.g. 0/1 indicator columns)
- Add a column of 1s for the intercept if desired (fit_intercept=True -- in sklearn.linear_model)

Target Array

The **target array** y is an array with elements that serve as "labels" for the n observations.

Target array y: This is our array of labels, i.e., response data that will be used to train the model. It may be categorical (0/1) as in classification problems and logistic regression, or have continuous numerical values as in linear regression.

How to convert columns from a dataframe to a features matrix X and a target array y?

Given a data frame containing both X and y data, we'll need to extract these two arrays from the data frame. We'll see a simple way to do this using Pandas.

3b: Two Main Types of Machine Learning

What is the main difference between supervised learning vs. unsupervised learning?

Supervised learning:

Nature of the Data You Have with Supervised Learning

When we have a target array y, its elements **label** the corresponding rows of X.

Types of Supervised Learning Labels

The two main types of labels are:

1. **Categorical (0/1) labels:** these correspond to **classification problems**, where the labels are said to **supervise** the modeling of how information in X can be used for training the classifier.

2. **Continuous numerical labels:** these correspond to **predictive regression problems** such as linear regresssion or regularized regresssion, where the goal is to predict numerical responses. The numerical y labels **supervise** training of the regression model, e.g., by minimizing and ordinary or penalized least squares criterion.

General Goal of Supervised Learning

Unsupervised learning

Nature of the Data You Have with Unsupervised Learning

General Goal of Unsupervised Learning

Common Types of Unsupervised Learning Algorithms

If we have unlabeled data, so y is not available, we can no longer classify the rows of X, however, it is often of interest to detect **clusters** of samples in X based on some assumptions about how subpopulations might separate. Applications include k-means clustering and mixtures of Gaussian distributions.

3c: General Modeling steps in the modeling/machine learning process

This section focuses on supervised model building, and, at the end, generalizing train/test methods to k-fold cross-validation.

Scikit-Learn are designed to faciliate a regular sequence of modeling steps:

Step 1: Choose the model class for the problem and import the Scikit-Learn estimator class

Step 2: Choose model hyperparameters to create an instance of the class (more later)

Step 3: Organize data into the features matrix X and target array y

<u>Step 4</u>: Train the model using the .fit() method

<u>Step 5</u>: Test the model on new or test data using the .predict() method

<u>Step 6</u>:Compute performance metrics, e.g., using the .score( ) method

For extensive further discussion, see VanDerPlas, Chapter 5. We cover the basics with some examples here.

Topic 4: Descriptive Analytics for Datasets with a Large Amount of Explanatory Variables (ie. High Dimensional Datasets)

Data Simulation Generation: with 20 features (explanatory variables) and binary target (response variable)

Import the necessary functions.

In [7]:
```
import numpy as np
import pandas as pd
from scipy.stats import norm, bernoulli
```

Generate our simulated data:

In [8]:

```
## make a coefficient vector for logit model
b0 = -1  # intercept
bvec = np.repeat([1,-1,0], [5, 5, 10])  # feature coefficients
n = 200
nX = bvec.size

Xmat = norm.rvs(size=n*nX, random_state=1).reshape((n, nX))
# Generate X and add the column names
##X = pd.DataFrame(
##    norm.rvs(size=n*nX, random_state=1).reshape((n, nX)),
##    columns=Xnames)
# Now generate the target array using X and the coefficient vector
odds = np.exp(b0 + np.matmul(Xmat, bvec)) # use matrix multiplication
gen_y = bernoulli.rvs(p=odds/(1+odds), size=n, random_state=12347)
# load X into data frame with names
Xnames = []
for i in range(nX):
    list.append(Xnames, 'X'+str(i+1))
gen_X = pd.DataFrame(Xmat, columns=Xnames)
display(gen_X.shape, gen_y.shape)
```

```
(200, 20)

(200,)
```

Dataframe version of our Feature Matrix

In [9]:

```
gen_X
```

Out[9]:

	X1	X2	X3	X4	X5	X6	X7	X8	
0	1.624345	-0.611756	-0.528172	-1.072969	0.865408	-2.301539	1.744812	-0.761207	0.
1	-1.100619	1.144724	0.901591	0.502494	0.900856	-0.683728	-0.122890	-0.935769	-0.
2	-0.191836	-0.887629	-0.747158	1.692455	0.050808	-0.636996	0.190915	2.100255	0.
3	-0.754398	1.252868	0.512930	-0.298093	0.488518	-0.075572	1.131629	1.519817	2.
4	-0.222328	-0.200758	0.186561	0.410052	0.198300	0.119009	-0.670662	0.377564	0.
...	...	...	...	...	...	...	...	...	
195	0.298112	-0.132637	-2.251077	2.893593	2.742155	-0.510218	0.833351	0.088922	0.
196	-0.284319	0.525410	-2.050876	0.211909	1.110508	0.751354	0.217867	-0.780517	0.
197	1.991515	1.299629	-0.592073	0.049659	2.478005	-1.778695	0.395753	2.003272	0.
198	0.874996	-1.049369	-0.607352	0.418138	-1.541780	-1.025194	0.325975	0.805144	0.
199	-0.138881	2.652140	-0.656247	0.279562	-0.607715	0.729814	-0.887188	0.077327	0.

200 rows × 20 columns

Dataframe Version of our Target Array

In [10]:
```python
gen_y
```

Out[10]:
```
array([0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0,
       0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
       1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0,
       1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
       0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1,
       1, 1])
```

Put it all together in df

Bundle into a data frame for the purpose of this example. Then we demonstrate how to extract the features matrix X and target y from a data frame in general, for any data set.

In [11]:
```python
df = gen_X
df['y'] = gen_y
df.head()
```

Out[11]:

	X1	X2	X3	X4	X5	X6	X7	X8	
0	1.624345	-0.611756	-0.528172	-1.072969	0.865408	-2.301539	1.744812	-0.761207	0.319
1	-1.100619	1.144724	0.901591	0.502494	0.900856	-0.683728	-0.122890	-0.935769	-0.267
2	-0.191836	-0.887629	-0.747158	1.692455	0.050808	-0.636996	0.190915	2.100255	0.120
3	-0.754398	1.252868	0.512930	-0.298093	0.488518	-0.075572	1.131629	1.519817	2.185
4	-0.222328	-0.200758	0.186561	0.410052	0.198300	0.119009	-0.670662	0.377564	0.121

5 rows × 21 columns

Now we have constructed our simulated data set, 'df', that we use to demonstrate the modeling methods in the rest of this section.

Given a data frame containing both features and target (X's and y's), how to extract X and y in a simple fashion?

In [12]:
```python
y = df['y']
X = df.drop(columns='y')
```

In [13]:
```
X.head()
```

Out[13]:

	X1	X2	X3	X4	X5	X6	X7	X8	
0	1.624345	-0.611756	-0.528172	-1.072969	0.865408	-2.301539	1.744812	-0.761207	0.319
1	-1.100619	1.144724	0.901591	0.502494	0.900856	-0.683728	-0.122890	-0.935769	-0.267
2	-0.191836	-0.887629	-0.747158	1.692455	0.050808	-0.636996	0.190915	2.100255	0.120
3	-0.754398	1.252868	0.512930	-0.298093	0.488518	-0.075572	1.131629	1.519817	2.185
4	-0.222328	-0.200758	0.186561	0.410052	0.198300	0.119009	-0.670662	0.377564	0.121

In [14]:
```
y
```

Out[14]:
```
0      0
1      1
2      0
3      0
4      0
      ..
195    1
196    0
197    1
198    1
199    1
Name: y, Length: 200, dtype: int32
```

Descriptive Analytics: For each of the 20 features, which feature does the y=1 group have the higher average?

Super slow way of doing this.

In [15]:
```
mean_matrix = np.array((X[y==0].mean(), X[y==1].mean()))
display(mean_matrix.shape, mean_matrix)
```

```
(2, 20)

array([[-2.22157664e-01, -2.04966538e-01, -2.43240376e-01,
        -3.63570421e-02, -1.93845863e-01,  9.36729581e-02,
         2.62709812e-01,  2.13169647e-01,  2.00029658e-01,
         4.63842261e-02,  7.77167483e-02, -6.72668629e-02,
         4.06244529e-04,  9.16813708e-02,  9.95053160e-02,
        -6.73780626e-02,  9.83761988e-02, -9.47603953e-02,
        -6.94209204e-02,  4.91980066e-02],
       [ 2.25164206e-01,  2.82792473e-01,  2.80512385e-01,
         4.03376804e-01,  1.78792096e-01, -4.00832721e-01,
        -1.61412650e-01, -1.31933518e-01, -8.48852940e-02,
        -4.39996214e-01,  7.77267095e-02,  2.42083521e-01,
         4.56481738e-02,  2.79365255e-02, -2.91626160e-01,
         4.88051708e-02,  3.90577553e-02, -1.18110230e-02,
         9.25214471e-02,  1.95490706e-01]])
```

Less slow way of doing this.

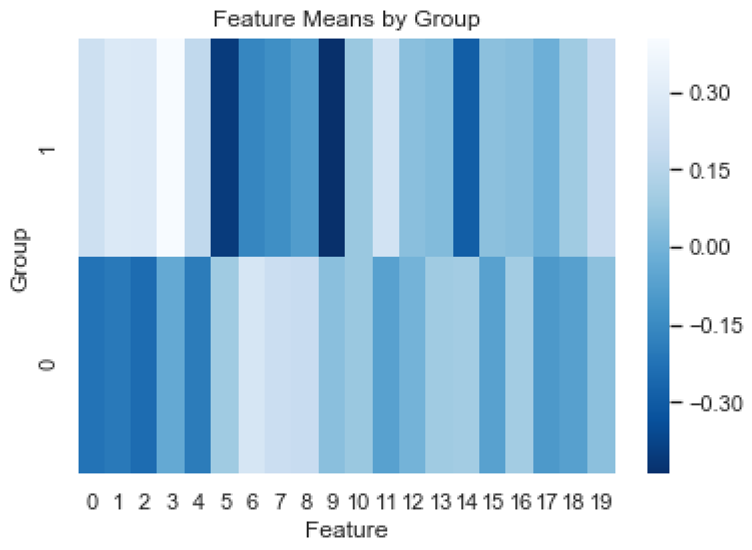
In [17]:
```
np.round(mean_matrix, 3)
```

Out[17]:
```
array([[-0.222, -0.205, -0.243, -0.036, -0.194,  0.094,  0.263,  0.213,
         0.2  ,  0.046,  0.078, -0.067,  0.   ,  0.092,  0.1  , -0.067,
         0.098, -0.095, -0.069,  0.049],
       [ 0.225,  0.283,  0.281,  0.403,  0.179, -0.401, -0.161, -0.132,
        -0.085, -0.44 ,  0.078,  0.242,  0.046,  0.028, -0.292,  0.049,
         0.039, -0.012,  0.093,  0.195]])
```

Faster (more visually appealing) way of doing this.

In [8]:
```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

In [9]:
```
fig = sns.heatmap(mean_matrix, annot=False, linewidths=0,
            square = False, cmap = 'Blues_r');
fig.set_ylim([0,2]);
plt.ylabel('Group');
plt.xlabel('Feature');
all_sample_title = 'Feature Means by Group'
plt.title(all_sample_title, size = 12);
plt.show(fig)
```



We can see that the two groups differ in their patterns of mean feature values. This is a reflection of how the different features contribute to the logit model through the coefficient vector weighting of features to generate 0/1 responses.

Topic 6: Non-regularized Regression Models

Two Main Purposes of Non-Regularized Regression Models

For Non-Regularized Linear or Logistic Regression, when should I use statmodels package vs. scikit-learn package?

In previous sections we used the Statmodels formula API to fit linear and logistic regression models. The advantage of the statsmodels API is in the amount of information it provides about model parameters, standard errors and hypothesis testing. Statsmodels is the current preferred choice if our goal is to make inferences about the model parameters.

As an alternative, Scikit-Learn provides machine learning tools focused on predictive analysis. The characteristics:

- Integration with machine learning tools for **cross-validation**;
- Routine implementation of **regularization** for modeling with high-dimensional feature matrices;

- Several choices for numerical optimizers for solving regularized regression problems.

Topic 7: (RECAP) How do we implement a penalty for the number of featues in a non-regularized regression model?

Regularization penalties automate feature selection to some extent, as an alternative to tracking AIC or BIC.

First, let's review what we do with non-regularized logisic regression.

Non-Regularized Logistic Regression: Objective Function

First consider "ordinary" **maximum likelihood estimation** for the logistic regression. We solve numerically for the coefficient estimates using a numerical optimization method such as a quasi-newton algorithm. The solver maximizes the log-likelihood, equivalently, minimizes twice the negative log likelihood:

1. What we do when fitting a SINGLE logistic regression model:

$$
\begin{aligned}
\text{Likelihood fit} = -2*llf &= -2*LLF(\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p) \\
&= -2\sum_{i=1}^{n}\{y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)\} \\
&\geq 0.
\end{aligned}
$$

2. After fitting MANY non-reguliarzed logistic regression models (with different combinations of explanatory variables), how do we select which model is best?

Using the *Optimal* Objective Function *Values* of SEVERAL Non-Reguliarzed Logistic Regression Models we MANUALLY Select the a Model that is Parsimonious

If we compare models with different numbers of coefficients using BIC, for example, we are attempting to minimize an overall criterion of the form:

$$-2 * llf + p * \log(n) = \text{likelihood fit} + \text{complexity penalty}$$

Topic 8: Automating Model Selection

IDEA: *How* can we automate this process so we don't have to fit multiple models? Aka: how can we make it so we only have to find the optimal solution to just ONE objective function?

IDEA: *Why* would we want to automate this process so we don't have to fit multiple models?

Topic 9: How can we modify the logistic regression objective function such that models having lots of features are penalized?

Three common methods of introducing penalties into the objective function

Regularization penalties combine the likelihood criterion and the penalty into one optimization criterion for estimating the feature coefficients. For a given level of complexity this can produce fitted models with better mean-square error predictive properties. Here a few choices that are currently in common use:

<u>Method 1</u>: L2 penalty (Ridge regression)

$$\text{Minimize: } (-2 * llf) + \lambda \sum_{j=1}^{p} \hat{\beta}_j^2$$

Shrinks parameters toward smaller absolute values to reduce variance. In linear regression this is called ridge regression. λ controls tradeoff between fit and shrinkage. It reduces the effect of correlation between features, which is an issue in high dimensions.

Using the ridge regression objective function, what types of optimal solutions for $\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p$ will minimize this function the most?

What would it mean if the optimal solution $(\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p)$ of the ridge regression objective function was one in which $\hat{\beta}_1 = 0, \hat{\beta}_2 = 0 \ldots, \hat{\beta}_p = 0$?

<u>Method 2</u>: L1 penalty (LASSO)

$$\text{Minimize: } (-2 * llf) + \lambda \sum_{j=1}^{p} |\hat{\beta}_j|$$

Using the ridge regression objective function, what types of optimal solutions for $\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p$ will minimize this function the most?

Which objective function will penalize "small" values of β_i more: LASSO objective function or ridge regression objective function?

Why would you want to use LASSO over ridge regression?

Produces sparse solutions (small number of nonzero parameters) by zeroing out some parameters. λ controls tradeoff between fit and sparsity. This method directly reduces redundant features.

Why would you want to use ridge regression over LASSO?

In LASSO and Ridge Regression, you select the value of λ *before* fitting the model. Selecting different values of λ may produce different optimal solutions. What types of solutions will LARGE values of λ prioritize for LASSo and Ridge Regression?

<u>Method 3</u>: Elastic Net Regression (Combination of LASSO and Ridge Regression)

Combines L1 and L2 penalization to produce sparse, low variance solutions. λ controls the tradeoff between fit and penalization. $\alpha \in [0, 1]$ controls the relative weight of L1 versus L2 penalization.

$$\text{Minimize: } (-2 * llf) + \lambda\left(\alpha \sum_{j=1}^{p} |\hat{\beta}_j| + \frac{1-\alpha}{2} \sum_{j=1}^{p} \hat{\beta}_j^2\right)$$

What types of solutions will LARGE values of λ prioritize for Elastic Net Regression?

What types of solutions will LARGE values of α prioritize for Elastic Net Regression?

What types of solutions will SMALL values of α prioritize for Elastic Net Regression?

The current default setting in the Scikit-Learn LogisticRegression module is L1 regularization. The function sets the degree of penalization using the input C, which is inversely related to λ. The default setting is C=1. Smaller values of C correspond to heavier regularization or more sparsity in the solution.

Topic 10: Simulation Example Continued: Compare Different Penalties

Using our simulated data with 20 features, let's compare for 4 different options for regularization.

1. Basic Logistic Regression (ie. no penalty)
2. Logistic Regression with L1 penalty (ie. LASSO Logistic Regression)
3. Logistic Regression with L2 penalty (ie. Ridge Logistic Regression)
4. Elastic Net Logistic Regression (Combination of L1 and L2 penalty)

In [18]:
```
from sklearn.linear_model import LogisticRegression
```

Fixing the Regularization Parameter

Fix the λ value.

In sklearn, $C = \frac{1}{\lambda}$.

In [19]:
```
C=0.3  # set the amount of penalization (1/lambda)
```

Running each of the Models

Basic Logistic Regression (no penalties)

- Using the 'newton-cg' solver. The newton-cg algorithm is a type of numerical analysis algorithm that goes about finding an optimal solution to a given objective function.
- This algorithm stops after 1000 iterations or when the algorithm has converged.
- The 'newton-cg' solver only works for: basic logistic regression and ridge regression.

In [20]:
```
clf0 = LogisticRegression('none', solver='newton-cg',
                          max_iter=1000)
clf0.fit(X,y)
```

Out[20]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='none',
                   random_state=None, solver='newton-cg', tol=0.0001, verbo
se=0,
                   warm_start=False)
```

LASSO Logistic Regression (L1 penalties)

- Using the 'liblinear' solver. liblinear is a tool that solves linear logistic regression optimization problems.
- This algorithm stops after 1000 iterations or when the algorithm has converged.
- The 'liblinear' solver only works for: LASSO logistic regression and logistic ridge regression.

In [21]:
```
clf1 = LogisticRegression('l1', solver='liblinear',
                                  max_iter=1000, C=C)
clf1.fit(X,y)
```

Out[21]:
```
LogisticRegression(C=0.3, class_weight=None, dual=False, fit_intercept=Tru
e,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='l1',
                   random_state=None, solver='liblinear', tol=0.0001, verbo
se=0,
                   warm_start=False)
```

Logistic Ridge Regression (L2 penalties)

- Using the 'liblinear' solver. liblinear is a tool that solves linear logistic regression optimization problems.
- This algorithm stops after 1000 iterations or when the algorithm has converged.
- The 'liblinear' solver only works for: LASSO logistic regression and logistic ridge regression.

In [13]:
```
clf2 = LogisticRegression('l2', solver='liblinear',
                                  max_iter=1000, C=C)
clf2.fit(X,y)
```

Out[13]:
```
LogisticRegression(C=0.3, class_weight=None, dual=False, fit_intercept=Tru
e,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbo
se=0,
                   warm_start=False)
```

Elastic Net Logistic Regression (Combination of L1 and L2 penalties)

- Using the 'saga' solver. saga is a numerical optimization method that only works for specific types of objective functions.
- This algorithm stops after 1000 iterations or when the algorithm has converged.
- The 'saga' solver only works for: elastic net logistic regression.
- The α in sklearn is represented as the "l1_ratio" parameter in the function. With an $\alpha =$ l1_ratio=0.7, this means that this particular elastic net model will favor solutions that more closely resemble:

In [14]:
```
clf3 = LogisticRegression('elasticnet', solver='saga',
                          max_iter=1000, l1_ratio=0.7, C=C)
clf3.fit(X,y)
```

Out[14]:
```
LogisticRegression(C=0.3, class_weight=None, dual=False, fit_intercept=Tru
e,
                   intercept_scaling=1, l1_ratio=0.7, max_iter=1000,
                   multi_class='warn', n_jobs=None, penalty='elasticnet',
                   random_state=None, solver='saga', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [15]:
```
#dir(clf3)
```

Now, let's extract the 20 resulting slope coefficients for each of the 4 models.

In [16]:
```
clf3.coef_
```

Out[16]:
```
array([[ 0.91339789,  0.84105838,  0.83298926,  0.65831549,  0.50126758,
        -0.87714052, -0.739675  , -0.73085726, -0.53128786, -0.97850072,
         0.        ,  0.17358884,  0.01481474, -0.13674186, -0.29364757,
         0.        ,  0.        ,  0.        ,  0.03755442,  0.06074195]])
```

In [17]:
```
clf3.intercept_
```

Out[17]:
```
array([-0.94797261])
```

In [18]:

```
dfcoef = pd.DataFrame(
    np.concatenate((clf0.coef_.T,
                    clf1.coef_.T,
                    clf2.coef_.T,
                    clf3.coef_.T),
                   axis=1)
)
dfcoef
```

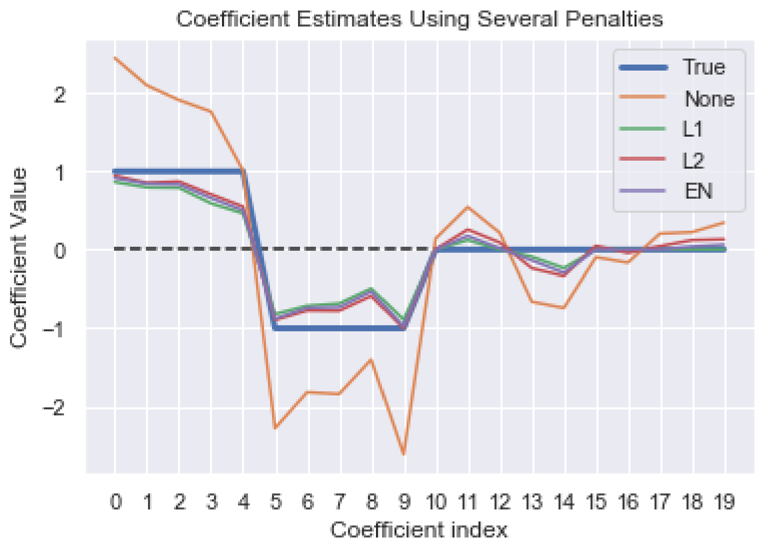
Out[18]:

	0	1	2	3
0	2.448386	0.862878	0.943341	0.913398
1	2.099151	0.794844	0.852618	0.841058
2	1.911414	0.789636	0.869266	0.832989
3	1.763950	0.590481	0.706093	0.658315
4	1.011388	0.462536	0.549192	0.501268
5	-2.277685	-0.818814	-0.893153	-0.877141
6	-1.817789	-0.713729	-0.773147	-0.739675
7	-1.840509	-0.684439	-0.776345	-0.730857
8	-1.400668	-0.493109	-0.588224	-0.531288
9	-2.610932	-0.890087	-1.001949	-0.978501
10	0.141495	0.000000	0.003993	0.000000
11	0.543221	0.121319	0.255975	0.173589
12	0.213494	0.000000	0.090158	0.014815
13	-0.661488	-0.090676	-0.236996	-0.136742
14	-0.741134	-0.235318	-0.333969	-0.293648
15	-0.097251	0.000000	0.047848	0.000000
16	-0.165984	0.000000	-0.042778	0.000000
17	0.204026	0.000000	0.042845	0.000000
18	0.222341	0.003096	0.120741	0.037554
19	0.344166	0.000000	0.131761	0.060742

Let's visualize the slope differences.

In [19]:

```python
plt.plot(dfcoef.index, bvec, lw=3)
for i in range(4):
    plt.plot(dfcoef.index, dfcoef[i])
plt.xticks(np.arange(0,20,1))
plt.xlabel('Coefficient index')
plt.ylabel('Coefficient Value')
plt.title('Coefficient Estimates Using Several Penalties')
plt.legend(['True', 'None','L1','L2','EN'], loc='upper right')
plt.hlines(y=0, xmin=0, xmax=19, linestyles='--')
plt.show()
```



Do the slope differences that we observed corroborate how we *expect* the different models to behave?

In [20]:
```
# demo the for loop
for i in range(4):
    print(i)
```

```
0
1
2
3
```

We see that the unregularized ML coefficients have the most variation, and appear to stray farther from the true population values than the regularized estimators. The penalized estimators are biased toward zero, which reduces variation amd also appears to improve their accruacy. Where the coefficients are close to zero, L1 and ElasticNet zero them out.

Note: C=1 is the default setting for the amount of penalization. Smaller values of C increase the amount of penalization. Here we used C=0.3

Topic 11: Regularized Logistic Regression with Train/Test split in Sci-Kit Learn

We illustrate using the simulated binary response data. We'll generate the feature matrix X and a binary target array y from a logistic regression model with 20 features.

First, let's train the a LASSO logistic regression model using a *single* training dataset and a *single* test dataset.

1. Import the model class:

In [21]:
```
from sklearn.linear_model import LogisticRegression
```

2. Create an instance of the model. We'll use an 'l1' penalty to regularize the regression for this example.

In [22]:
```
logitReg = LogisticRegression(penalty='l1',
                              solver='liblinear',
                              C=1,
                              max_iter=1000)
```

3. Train the model (for now we'll use a 1-fold split... aka what we've done in the past)

In [23]:
```
from sklearn.model_selection import train_test_split
```

```
In [24]: X_train, X_test, y_train, y_test = train_test_split(
             X, y, test_size=0.20, random_state=42)
```

```
In [25]: logitReg.fit(X_train, y_train)

Out[25]: LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
                            intercept_scaling=1, l1_ratio=None, max_iter=1000,
                            multi_class='warn', n_jobs=None, penalty='l1',
                            random_state=None, solver='liblinear', tol=0.0001, verbo
         se=0,
                            warm_start=False)
```

4. Get predictions for test data

```
In [26]: # category predictions using 0.5 threshold
         yhat = logitReg.predict(X_test)
```

```
In [27]: yhat

Out[27]: array([0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0,
                0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0])
```

```
In [28]: # (0,1) predictive probabilities
         phat01 = logitReg.predict_proba(X_test)
```

```
In [29]: phat01[0:5]

Out[29]: array([[0.6932936 , 0.3067064 ],
                [0.97686061, 0.02313939],
                [0.98620941, 0.01379059],
                [0.00512146, 0.99487854],
                [0.90071749, 0.09928251]])
```

```
In [30]: phat1 = phat01[:,1]
         phat1

Out[30]: array([3.06706403e-01, 2.31393948e-02, 1.37905907e-02, 9.94878540e-01,
                9.92825058e-02, 1.90376145e-02, 6.57872625e-03, 2.23362678e-03,
                3.80080428e-01, 1.41567279e-01, 6.79882444e-01, 1.48639985e-02,
                3.65025946e-02, 9.92545221e-01, 9.44923312e-01, 4.64036069e-03,
                2.43577570e-01, 6.92031058e-01, 2.82355052e-01, 1.35650095e-02,
                3.97975802e-04, 6.97796832e-04, 2.53443624e-01, 9.99058499e-01,
                8.17926320e-01, 4.59442716e-04, 6.46822475e-01, 8.87206409e-01,
                2.39646059e-01, 4.00899276e-02, 2.31581710e-01, 9.61382651e-02,
                7.82486981e-01, 7.57412455e-01, 4.87865283e-01, 8.57162460e-01,
                3.27152350e-03, 2.50055945e-01, 1.08285135e-01, 3.59258526e-01])
```

5. Evaluate the results

Accuracy

In [31]:
```
# Accuracy:
score = logitReg.score(X_test, y_test)
score
```
Out[31]:
```
0.825
```

Confusion matrix

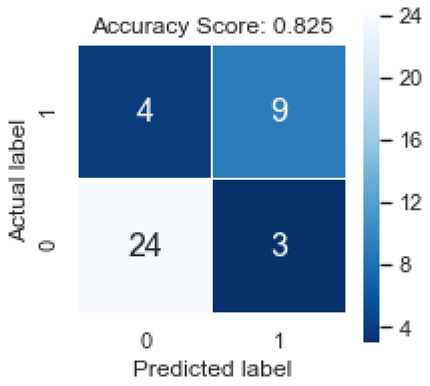
In [32]:
```
# Confusion matrix
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn import metrics
```

In [33]:
```
# rows are actual labels (0, 1), columns are predicted label (0,1)
cm = metrics.confusion_matrix(y_test, yhat)
cm
```
Out[33]:
```
array([[24,  3],
       [ 4,  9]])
```

Confusion matrix visualization using heatmap

Code adapted from: (Galarnyk, 2017 (https://towardsdatascience.com/logistic-regression-using-python-sklearn-numpy-mnist-handwriting-recognition-matplotlib-a6b31e2b166a))

In [34]:
```python
plt.figure(figsize=(3,3))
fig = sns.heatmap(cm, annot=True, annot_kws={"size": 16},
                  square=True,
                  linewidths=0.5,
                  cmap = 'Blues_r')
fig.set_ylim([0,2])
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
all_sample_title = 'Accuracy Score: {0}'.format(score)
plt.title(all_sample_title, size = 12)
plt.show(fig)
```



Here the cells are arranged so the lower left (0,0) and upper right (1,1) are the correct classfication counts for true negatives and true positives, respectively.

Sensitivity and specificity

In [35]:
```python
def senspec(y, score, thresh, index=0):
    yhat = 1*(score >= thresh)
    tn, fp, fn, tp = metrics.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 [36]:
```python
senspec(y_test, phat1, 0.5)   # 0.5 threshold
```

Out[36]:

	tn	fp	fn	tp	sens	spec	accuracy
0	24	3	4	9	0.692308	0.888889	0.825

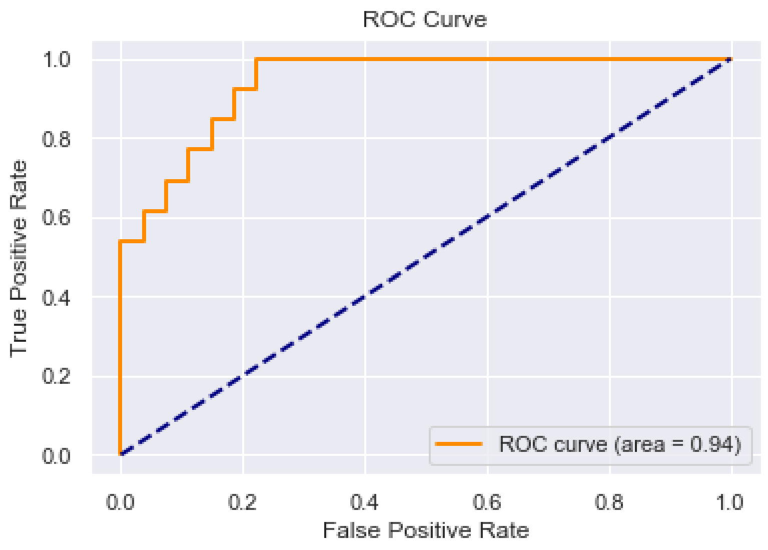
In [37]:
```python
senspec(y_test, phat1, 0.33)  # 0.33 threshold
```
Out[37]:
```
   tn  fp  fn  tp      sens      spec  accuracy
0  23   4   2  11  0.846154  0.851852      0.85
```

ROC curve

In [38]:
```python
fpr, tpr, score = metrics.roc_curve(y_true=y_test, y_score=phat1)
auc = metrics.roc_auc_score(y_true=y_test, y_score=phat1)
```

In [39]:
```python
def plot_roc(fpr, tpr, auc, lw=2):
    plt.plot(fpr, tpr, color='darkorange', lw=lw,
             label='ROC curve (area = '+str(round(auc,3))+')')
    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc="lower right")
    plt.show()
```

In [40]:
```python
plot_roc(fpr, tpr, auc)
```



Next let's train each of the four regularization types (ie. none, L1, L2, and elastic net) Using Cross Validation Scoring.

In cross validation, we cycle through all the k-fold train/test splits of the data using each 1/k subset as the test set with the other 1 - 1/k as the training set. In this way we obtain k different splits of the data. We average over the k splits. The variation in performance scores can also gives us a standard error on the performance measure.

In [41]:
```python
from sklearn.model_selection import cross_val_score
```

Without regularization

In [42]:
```
modclass0 = LogisticRegression(penalty='none', solver='newton-cg')
scores0 = cross_val_score(modclass0, X, y, cv=5)
scores0
```
Out[42]:
```
array([0.825, 0.675, 0.8  , 0.85 , 0.9  ])
```

In [43]:
```
print("Accuracy: %0.2f (+/- %0.2f)" % (scores0.mean(), 2*scores0.std()/n
```
```
Accuracy: 0.81 (+/- 0.07)
```

Using L1 regularization

In [44]:
```
modclass1 = LogisticRegression(penalty='l1',
                               solver='saga',
                               C=0.3,
                               max_iter=1000)
scores1 = cross_val_score(modclass1, X, y, cv=5)
scores1
```
Out[44]:
```
array([0.825, 0.75 , 0.8  , 0.8  , 0.85 ])
```

In [45]:
```
print("Accuracy: %0.2f (+/- %0.2f)" % \
      (scores1.mean(), 2*scores1.std()/np.sqrt(5)))
```
```
Accuracy: 0.80 (+/- 0.03)
```

Using L2 regularization

In [46]:
```
modclass2 = LogisticRegression(penalty='l2',
                               solver='saga',
                               C=0.3,
                               max_iter=1000)
scores2 = cross_val_score(modclass2, X, y, cv=5)
scores2
```
Out[46]:
```
array([0.825, 0.75 , 0.8  , 0.85 , 0.9  ])
```

In [47]:
```
print("Accuracy: %0.2f (+/- %0.2f)" % \
      (scores2.mean(), 2*scores2.std()/np.sqrt(5)))
```
```
Accuracy: 0.82 (+/- 0.04)
```

Using Elastic Net regularization

In [48]:

```python
modclass3 = LogisticRegression(penalty='elasticnet',
                               solver='saga',
                               l1_ratio=0.75,
                               C=0.3,
                               max_iter=1000)
scores3 = cross_val_score(modclass3, X, y, cv=5)
scores3
```

Out[48]:

```
array([0.825, 0.75 , 0.825, 0.85 , 0.85 ])
```

In [49]:

```python
print("Accuracy: %0.2f (+/- %0.2f)" % \
      (scores3.mean(), 2*scores3.std()/np.sqrt(5)))
```

```
Accuracy: 0.82 (+/- 0.03)
```

STAT 207, Victoria Ellison and Douglas Simpson, University of Illinois at Urbana-Champaign

In [ ]:

```python

```