Algorithm 1 - Logistic Regression Classifier

In [1]:
```
%matplotlib inline

import os
import sys
import json

import numpy as np
import pandas as pd

from matplotlib import pyplot as plt
from IPython.display import display
import seaborn as sns
sns.set()

# Add our local functions to the path
sys.path.append(os.path.join(os.getcwd(), 'src'))
from models import evaluation
from data.load_data import (get_country_filepaths,
                            split_features_labels_weights,
                            load_data)
from features import process_features
from features.process_features import get_vif, standardize

ALGORITHM_NAME = 'lr'
COUNTRY = 'riau'
TRAIN_PATH, TEST_PATH = get_country_filepaths(COUNTRY)
```

In [2]:
```
# Load training data
X_train, y_train, w_train = split_features_labels_weights(TRAIN_PATH)

# summarize loaded data
print('Data has {:,} rows and {:,} columns' \
        .format(*X_train.shape))

print('Percent poor: {:0.1%} \tPercent non-poor: {:0.1%}' \
        .format(*y_train.miskin.value_counts(normalize=True, ascending=True)))

# print first 5 rows of data
X_train.head()
```

```
Data has 6,136 rows and 506 columns
Percent poor: 4.9%      Percent non-poor: 95.1%
```

Out[2]:

	r105	r1701	r1702	r1703	r1704	r1705	r1706	r1707	r1708	r1801	...	kons_307	kons_308	kons_309	kons_310	der_nchild10under	der_nmalesover10	der_
0	1	5	5	5	5	5	5	5	5	0	...	0	0	0	0	0	0	
1	2	5	5	5	5	5	5	5	5	1	...	0	0	0	0	0	3	
2	1	5	5	5	5	5	5	5	5	1	...	0	0	0	0	0	3	
3	2	5	5	5	5	5	5	5	5	1	...	0	0	0	0	1	3	
4	2	5	5	5	5	5	5	5	5	1	...	0	0	0	0	0	1	

5 rows × 506 columns

Unbalanced vs Balanced Datasets

Class labels in the Riau data are unbalanced. This means that there are much fewer examples of "poor" households ~~(6%) than "non-poor" (~~94%). If we were only using classification accuracy as a metric (The number of correct predictions), this means that we could simply predict that every household is non-poor and have a model with ~94% accuracy! However, this would obviously not help us reach our goals for actually understanding and predicting poverty in Riau.

As a quick demonstration of this, let's simulate a dummy classifier that predicts '0' with a 50% probability for everything:

In [3]:
```
# Load the test set
X_test, y_test, w_test = split_features_labels_weights(TEST_PATH)

# Predict everything as 'non-poor', with 50% probability
y_pred = np.zeros(len(y_test))
y_prob = np.ones(len(y_test)) * 0.5

# Evaluate performance
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, show=True)
```

```
C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\sklearn\metrics\_classification.py:1245: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Model Scores

	score
accuracy	0.950635
recall	0.000000
precision	0.000000
f1	0.000000
cross_entropy	0.693147
roc_auc	0.500000
cohen_kappa	0.000000

As we predicted, the classifier gives us an accuracy of ~95%, but we can see pretty clearly from the other metrics that it is not performing well. This is one reason why accuracy is not the only metric we use to evaluate the performance of a model. In the case of poverty prediction, we may be more concerned with having a high recall, which in this case is the fraction of 'poor' households we predict correctly as 'poor' over the total number of actual 'poor' households.

Now let's see how an actual LogisticRegression classifier is affected by unbalanced data. We'll start with a small dataset of just a few features and see if the same multicollinearity issues exist.

Dummy variables

Another consideration is the creation of dummy variables. Some classification algorithms are able to handle categorical variables easily, but others require the inputs to be numeric. When we have categorical variables, the preffered method is to create dummy variables. This takes the categorical feature and creates a new binary column for each value. We can also include a dummy to deal with missing values. We don't necessarily want a dummy for every column, though. If we have n columns from n categories, every column is actually a linear combination of the other columns, creating a multicollinearity problem known as the dummy variable trap. To deal with this issue, we drop the first dummy variable for each categorical variable. Pandas has a nice function called get_dummies() that makes this process simple.

We will also want to remove features that are not useful for a classification problem, such as empty or constant columns and duplicate columns.

In [4]:
```
X_train = pd.get_dummies(X_train, drop_first=True, dummy_na=True, prefix_sep='__')
X_test = pd.get_dummies(X_test, drop_first=True, dummy_na=True, prefix_sep='__')

print("X_train shape with dummy variables added", X_train.shape)
print("X_test shape with dummy variables added", X_test.shape)
```

```
X_train shape with dummy variables added (6136, 506)
X_test shape with dummy variables added (2046, 506)
```

In [5]:
```
# remove columns with only one unique value (all nan dummies from columns with no missing values)

X_train = X_train.loc[:, X_train.nunique(axis=0) > 1]
X_test = X_test.loc[:, X_test.nunique(axis=0) > 1]

print("X_train shape with constant columns dropped", X_train.shape)
print("X_test shape with constant columns dropped", X_test.shape)
```

```
X_train shape with constant columns dropped (6136, 503)
X_test shape with constant columns dropped (2046, 503)
```

In [6]:
```
# remove duplicate columns - these end up being all from nan or Not Applicable dummies

process_features.drop_duplicate_columns(X_train, ignore=['fwt', 'weind'], inplace=True)
process_features.drop_duplicate_columns(X_test, ignore=['fwt', 'weind'], inplace=True)

print("X_train shape with duplicate columns dropped", X_train.shape)
print("X_test shape with duplicate columns dropped", X_test.shape)
```

```
X_train shape with duplicate columns dropped (6136, 497)
X_test shape with duplicate columns dropped (2046, 500)
```

In [7]:
```
# X_train.to_csv("X_train_with_dummy.csv", index=False, header=True)
```

Simple Model

In [8]:
```
# Select a few columns for this example
selected_columns = [
    'r301',
    'der_nchild10under',
    'der_nmalesover10',
    'der_nfemalesover10',
    'der_nliterate',
    'der_nbekerja',
    'r1809a',
    'r1816',
    'kons_305',
    'kons_262'
]

print("X shape with selected columns:", X_train[selected_columns].shape)
```

```
X shape with selected columns: (6136, 10)
```

In [9]:
```
get_vif(X_train[selected_columns])
```

```
C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\statsmodels\stats\outliers_influence.py:193: RuntimeWarning: divide by zero encountered in double_scalars
  vif = 1. / (1. - r_squared_i)
```

Out[9]:
```
r301                        inf
der_nchild10under           inf
der_nmalesover10            inf
der_nfemalesover10          inf
der_nliterate         68.733254
der_nbekerja           6.218668
r1809a                 2.331949
r1816                  3.485068
kons_305               1.041615
kons_262               1.027109
Name: variance_inflaction_factor, dtype: float64
```

Several of these VIF results are very high, so let's standardize the numeric data and check again.

In [10]:
```
standardize(X_train)
get_vif(X_train[selected_columns])
```

```
C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\statsmodels\stats\outliers_influence.py:193: RuntimeWarning: divide by zero encountered in double_scalars
  vif = 1. / (1. - r_squared_i)
```

Out[10]:
```
r301                        inf
der_nchild10under           inf
der_nmalesover10            inf
der_nfemalesover10          inf
der_nliterate         10.376394
der_nbekerja           1.443363
r1809a                 1.034608
r1816                  1.036954
kons_305               1.013604
kons_262               1.003865
Name: variance_inflaction_factor, dtype: float64
```

These VIF results are still very high, so once again let's remove the r301 (household size) feature

In [11]:
```python
selected_columns.remove('r301')
print(selected_columns)

get_vif(X_train[selected_columns])
```

```
['der_nchild10under', 'der_nmalesover10', 'der_nfemalesover10', 'der_nliterate', 'der_nbekerja', 'r1809a', 'r1816', 'kons_305',
'kons_262']
```

Out[11]:
```
der_nchild10under       1.652763
der_nmalesover10        5.557242
der_nfemalesover10      4.588654
der_nliterate          10.376394
der_nbekerja            1.443363
r1809a                  1.034608
r1816                   1.036954
kons_305                1.013604
kons_262                1.003865
Name: variance_inflaction_factor, dtype: float64
```

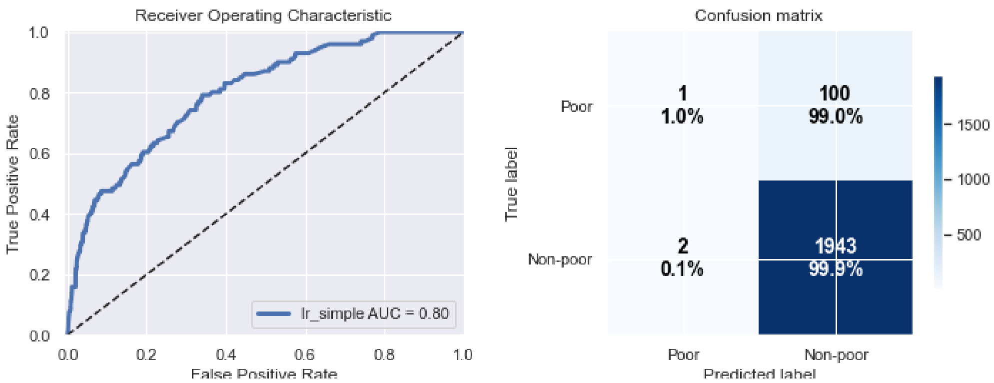
Now the VIF results are back in an acceptable range, so let's use these features and train a LogisticRegression model. We can use the load_data function in the load_data.py module, which uses our standardize function by default.

We'll also store these selected features in RIAU_SIMPLE_FEATURES so we can use the same subset in other notebooks.

In [12]:
```python
# Same method for getting the coefficients
def get_coefs_df(X, coefs, index=None):
    coefs_df = pd.DataFrame(np.std(X, 0)*coefs)
    coefs_df.columns = ["coef_std"]
    coefs_df['coef'] = coefs
    coefs_df['abs'] = coefs_df.coef_std.apply(abs)
    if index is not None:
        coefs_df.index = index
    return coefs_df
```

In [13]:
```python
from sklearn.linear_model import LogisticRegression

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH, selected_columns=selected_columns)

# Fit the model
model = LogisticRegression()
%time model.fit(X_train, y_train)

# Get an initial score
%time score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))

# Store coefficients
coefs = get_coefs_df(X_train, model.coef_[0])

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH, selected_columns=selected_columns)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob,
                                    store_model=True,
                                    model_name='simple',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
Wall time: 28 ms
Wall time: 4 ms
In-sample score: 95.06%
```



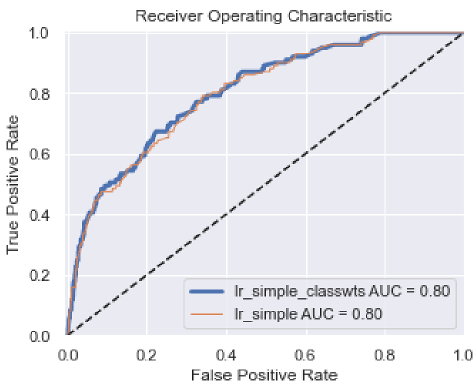
Here we see that the recall is only ~1%, which means we do a very poor job of predicting that poor households are poor. This is probably due to the fact that we have so few examples of poor households to train the model on. An interesting metric to consider is the Cohen's Kappa metric. This normalizes the classification accuracy by the imbalance of the classes in the data. Here we can see it is only ~1,6%.

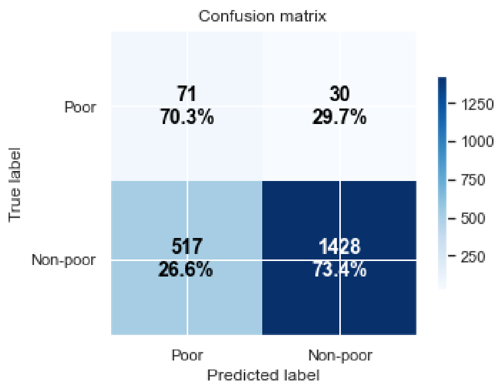
Class Weighting

Scikit-Learn offers several methods to deal with unbalanced classes. One is to adjust the weights of the classes to be inversely proportional to the class frequencies. This can be a simple way to increase the recall of the model, but usually has a negative effect on the accuracy and precision.

In [14]:
```
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH, selected_columns=selected_columns)

# Fit the model using class_weight='balanced'
model = LogisticRegression(class_weight='balanced')
%time model.fit(X_train, y_train)

# Get an initial score
%time score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))

# Store coefficients
coefs = get_coefs_df(X_train[selected_columns], model.coef_[0])

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH, selected_columns=selected_columns)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob,
                                    compare_models='lr_simple',
                                    store_model=True,
                                    model_name='simple_classwts',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
Wall time: 14 ms
Wall time: 2 ms
In-sample score: 73.19%
```





Model Scores

	lr_simple_classwts	lr_simple
accuracy	0.732649	0.950147
recall	0.702970	0.009901
precision	0.120748	0.333333
f1	0.206096	0.019231
cross_entropy	0.537430	0.164615
roc_auc	0.804167	0.799743
cohen_kappa	0.133050	0.016430

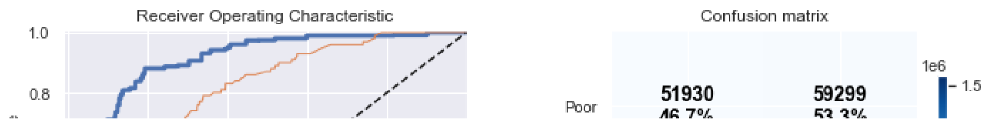
```
Actual poverty rate: 6.82%
Predicted poverty rate: 41.00%
```

This did not change the AUC much, but it made a significant improvement to the model's recall, bringing it up to about 70%.

Now let's try doing this with the full feature set with sample weights and see the effects:

In [15]:
```python
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)

# Fit the model
model = LogisticRegression()
%time model.fit(X_train, y_train, sample_weight=w_train)

# Get an initial score
%time score = model.score(X_train, y_train, sample_weight=w_train)
print("In-sample score: {:0.2%}".format(score))
coefs = get_coefs_df(X_train, model.coef_[0])['abs']

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, w_test,
                                    compare_models='lr_simple',
                                    store_model=True,
                                    model_name='full',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\sklearn\linear_model\_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(

Wall time: 504 ms
Wall time: 20 ms
In-sample score: 99.91%
```



This is much better than with the smaller feature set, but we still only have a recall of ~46.7%, which is not very good. Let's try using the balanced class_weights and see how it improves

In [16]:
```python
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)

# Fit the model with class_weight='balanced'
model = LogisticRegression(class_weight='balanced')
%time model.fit(X_train, y_train, sample_weight=w_train)

# Get an initial score
%time score = model.score(X_train, y_train, sample_weight=w_train)
print("In-sample score: {:0.2%}".format(score))
coefs = get_coefs_df(X_train, model.coef_[0])['abs']

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance and store model
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, w_test,
                                    compare_models='lr_full',
                                    store_model=True,
                                    model_name='full_classwts',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

False Positive Rate

Predicted label

Model Scores

	lr_full_classwts	lr_full
accuracy	0.929499	0.930738
recall	0.479928	0.466875
precision	0.450620	0.457938
f1	0.464812	0.462363
cross_entropy	1.082523	1.327124
roc_auc	0.914168	0.904081
cohen_kappa	0.400154	0.363980

```
Actual poverty rate: 6.82%
Predicted poverty rate: 7.29%
```

Oversampling and Undersampling

Oversampling and Undersampling Another method is to resample the dataset so the classes are balanced. This can be done by either oversampling or undersampling. With oversampling, we randomly replicate samples of the under-represented class. This is typically the preferred method when the dataset is rather small, on the order of a few thousand records. Undersampling reduces the size of the dataset by sampling the over-represented class. This is a prefereable method when the dataset is very large, since reducing the size of the training set can also reduce the computational cost.

The Riau dataset contains about 8,182 records, with only 6.82% being in the 'poor' class. If we use undersampling, this will reduce our training set to ~ records. If we use oversampling, we will increase the size of the dataset to about ~ records. We will try both approaches here and see which offers better performance.

Fortunately, there is a Python package called imbalanced-learn that provides implementations of several popular oversampling and undersampling techniques and is compatible with scikit-learn.

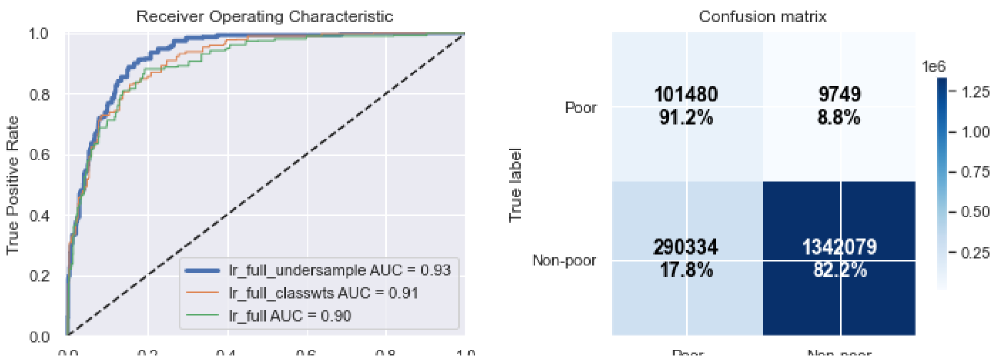
Undersampling

We'll apply undersampling using the RandomUnderSampler function from imbalanced-learn. This randomly takes samples the majority class to match the number of records from the under-represented class (or to reach a desired class ratio).

As a note, the imblearn functions return an array rather than a dataframe, so we'll need to store the column names if we want to inspect features or coefficients later.

In [17]:
```
from imblearn.under_sampling import RandomUnderSampler

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns

# Apply random undersampling
X_train, y_train = RandomUnderSampler().fit_resample(X_train, y_train)
print("X shape after undersampling: ", X_train.shape)

# Fit the model
model = LogisticRegression()
%time model.fit(X_train, y_train)

# Get an initial score
%time score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))

# Store coefficients
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance and store model
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, w_test,
                                    compare_models=['lr_full_classwts',
                                                    'lr_full'],
                                    store_model=True,
                                    model_name='full_undersample',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
X shape after undersampling:  (602, 506)
Wall time: 49 ms
Wall time: 7 ms
In-sample score: 100.00%
```



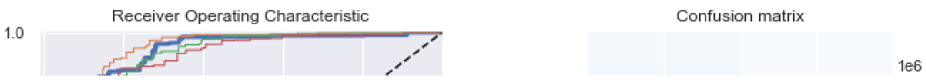
This gives us a slightly better recall than using class weights. It's also considerably faster, since it reduces the size of the training set to less than 10,000 records.

Oversampling

Next, we'll apply oversampling. One of the most popular oversampling methods is called SMOTE, or Synthetic Minority Oversampling Technique. This works by creating synthetic samples of the under-represented class by finding nearest neighbors and making minor random perturbations.

In [18]:
```python
from imblearn.over_sampling import SMOTE

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_resample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# Fit the model
model = LogisticRegression()
%time model.fit(X_train, y_train)

# Get an initial score
%time score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))

# Store coefficients
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance and store model
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, w_test,
                                    compare_models=['lr_full_undersample',
                                                    'lr_full_classwts',
                                                    'lr_full'],
                                    store_model=True,
                                    model_name='full_oversample',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
X shape after oversampling:  (11670, 506)

C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\sklearn\linear_model\_logistic.py:763: ConvergenceWarning: lbfgs fail
ed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/lin
ear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(

Wall time: 781 ms
Wall time: 33 ms
In-sample score: 99.17%
```



This gives us better results than undersampling and weighted classes, so we will use this method going forward. Note that it increases the size of the training set, but it appears to be more computationally efficient than using class weights.

Cross Validation and Parameter Tuning

Now that we have a good method for dealing with the unbalanced dataset, let's also apply cross-validation and some hyperparameter tuning using the LogisticRegressionCV model.

In [19]:
```
from sklearn.linear_model import LogisticRegressionCV

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_resample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# Fit the model
model = LogisticRegressionCV(Cs=10, cv=5, verbose=1)
%time model.fit(X_train, y_train)

# Get an initial score
%time score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)

# Display best parameters
print("Best model parameters: C={}".format(model.C_[0]))

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance and store model
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, w_test,
                                    compare_models=['lr_full_oversample',
                                                    'lr_full_undersample',
                                                    'lr_full_classwts',
                                                    'lr_full'],
                                    store_model=True,
                                    model_name='full_oversample_cv',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/lin
ear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\sklearn\linear_model\_logistic.py:763: ConvergenceWarning: lbfgs fail
ed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/lin
ear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
C:\ProgramData\Anaconda3\envs\satudata\lib\site-packages\sklearn\linear_model\_logistic.py:763: ConvergenceWarning: lbfgs fail
ed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

Using cross-validation and tuning the C parameter is much more computationally expensive to train the model. We may be able to do better by performing more detailed feature selection and fine tuning the model further, but this serves as a decent baseline model for the Riau dataset for this project.

Feature selection

let's pick a subset of features using the 'l1' regularization and see how the model performs using only this subset.

In [20]:
```
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_resample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# Fit the model
model = LogisticRegressionCV(cv=5, penalty='l1', Cs=[2e-3] , solver='liblinear')
%time model.fit(X_train, y_train)
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)
coefs = coefs[coefs.coef != 0]
print("{} features selected".format(coefs.shape[0]))
display(coefs)
feats = coefs.index.values
```

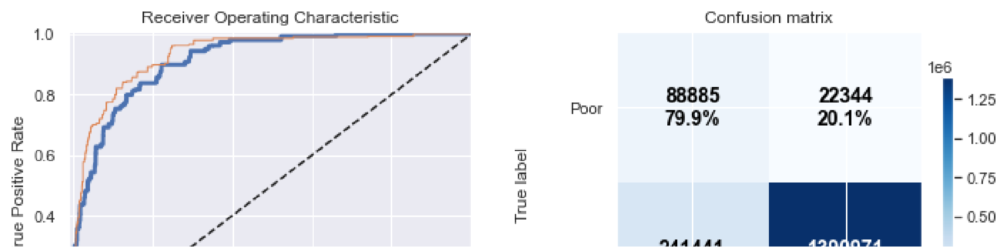
```
X shape after oversampling:  (11670, 506)
Wall time: 1.71 s
30 features selected
```

	coef_std	coef	abs
r1702	-0.020237	-0.016706	0.020237
r1703	-0.099552	-0.079181	0.099552
r1707	-0.009562	-0.006482	0.009562
r1808	0.041802	0.048638	0.041802
r1817	0.159701	0.124832	0.159701
r2001b	0.272981	0.271010	0.272981
r2207f2	0.000671	0.000509	0.000671
r301	0.694164	0.676498	0.694164
kons_6	-0.039085	-0.046599	0.039085
kons_56	-0.103645	-0.110045	0.103645
kons_68	-0.045331	-0.053570	0.045331
kons_81	-0.014798	-0.017493	0.014798
kons_107	-0.099839	-0.116881	0.099839
kons_111	-0.032722	-0.037553	0.032722
kons_140	-0.004301	-0.004823	0.004301
kons_160	-0.029058	-0.033263	0.029058
kons_166	-0.165489	-0.180331	0.165489
kons_179	-0.028771	-0.031588	0.028771
kons_193	0.040721	0.035816	0.040721
kons_195	-0.053219	-0.059341	0.053219
kons_207	-0.001633	-0.001329	0.001633
kons_214	-0.187117	-0.164604	0.187117
kons_224	0.019453	0.017011	0.019453
kons_229	-0.263362	-0.261414	0.263362
kons_238	-0.044919	-0.046364	0.044919
kons_268	-0.075897	-0.075838	0.075897
kons_277	-0.004414	-0.004541	0.004414
kons_278	-0.059185	-0.064101	0.059185
der_nchild10under	0.213294	0.196117	0.213294
der_ninternetpast3mo	-0.120950	-0.125674	0.120950

Now let's see how the model performs with this subset of features

In [21]:
```python
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH, selected_columns=feats)
cols = X_train.columns

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_resample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# Fit the model
model = LogisticRegressionCV(Cs=10, cv=5, verbose=1)
%time model.fit(X_train, y_train)

# Get an initial score
%time score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)

# Display best parameters
print("Best model parameters: C={}".format(model.C_[0]))

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH, selected_columns=feats)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

# Evaluate performance and store model
metrics = evaluation.evaluate_model(y_test, y_pred, y_prob, w_test,
                                    compare_models='lr_full_oversample_cv',
                                    store_model=True,
                                    model_name='l1_feats_oversample_cv',
                                    prefix=ALGORITHM_NAME,
                                    country=COUNTRY,
                                    model=model,
                                    features=coefs)
```

```
X shape after oversampling:  (11670, 30)

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    1.0s finished

Wall time: 1.11 s
Wall time: 3 ms
In-sample score: 91.52%
Best model parameters: C=2.782559402207126
```



Using this method, we get slightly worse performance than the full feature model, but we have reduced the number of features to less than 100.

Let's inspect the coefficients for the features we selected and look at the consumable items that remained in the model.

In [22]:
```python
cons_feats = [x for x in feats if x[0:5] == 'kons_']
print("{} consumables features selected:".format(len(cons_feats)))
for x in cons_feats:
    print(x)
```

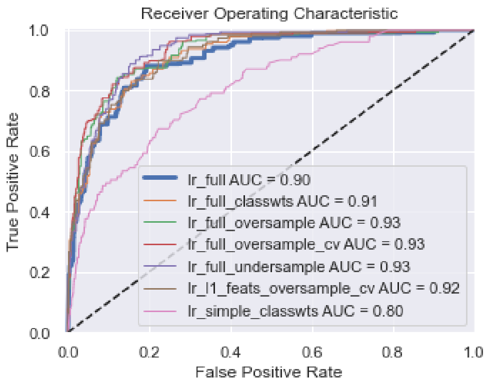
```
20 consumables features selected:
kons_6
kons_56
kons_68
kons_81
kons_107
kons_111
kons_140
kons_160
kons_166
kons_179
kons_193
kons_195
kons_207
kons_214
kons_224
kons_229
kons_238
kons_268
```

Logistic Regression Riau Summary

In this notebook, we demonstrated applying a logistic regression classifier to an unbalanced dataset. We introduced methods to deal with unbalanced classes such as SMOTE for oversampling, and highlighted the impact this has on how we evaluate a model.

We will use the results of the logistic regression classifier as a baseline for the other algorithms we will consider. In the following notebooks, we will introduce some new concepts but will primarily focus on the unique characteristics of each classifier model.

In [23]:
```
evaluation.compare_algorithm_models(ALGORITHM_NAME, COUNTRY)
```



Model Scores

	accuracy	recall	precision	f1	cross_entropy	roc_auc	cohen_kappa	pov_rate_error
lr_full	0.930738	0.466875	0.457938	0.462363	1.327124	0.904081	0.363980	0.000409
lr_full_classwts	0.929499	0.479928	0.450620	0.464812	1.082523	0.914168	0.400154	0.004715
lr_full_oversample	0.936187	0.639757	0.499863	0.561224	0.384259	0.926629	0.461351	0.017743
lr_full_oversample_cv	0.938166	0.693797	0.511302	0.588731	0.280809	0.933382	0.493789	0.023169
lr_full_undersample	0.827899	0.912354	0.259000	0.403465	0.709046	0.931591	0.305514	0.162158
lr_l1_feats_oversample_cv	0.848716	0.799117	0.269082	0.402599	0.367670	0.915135	0.334915	0.121121
lr_simple_classwts	0.732649	0.702970	0.120748	0.206096	0.537430	0.804167	0.133050	0.341761