Detecting anomalies in credit card transaction data (Python, scikit-learn)

Author: [Dennis W. Hallema (https://www.linkedin.com/in/dennishallema)](https://www.linkedin.com/in/dennishallema)

Description: Supervised classification procedure for detecting fraudulous credit card transactions in a large dataset. This procedure compares the performance of four classifiers: Logistic Regression, Kernel Support Vector Classifier, Stochastic Gradient Boosting and Random Forest.

Dependencies: See `environment.yml` .

Data: PCA transformed credit card transaction data collected in Europe over the course of two days. This anonymized dataset was created by Worldline and the Machine Learning Group of Université Libre de Bruxelles ([http://mlg.ulb.ac.be (http://mlg.ulb.ac.be)](http://mlg.ulb.ac.be)).

Disclaimer: Use at your own risk. No responsibility is assumed for a user's application of these materials or related materials.

References:

- Dal Pozzolo, A., Caelen, O., Le Borgne, Y-A, Waterschoot, S. & Bontempi, G. (2014). Learned lessons in credit card fraud detection from a practitioner perspective. Expert Systems with Applications, 41(10), 4915-4928.
- Dal Pozzolo, A., Boracchi, G., Caelen, O., Alippi, C. & Bontempi, G. (2018). Credit card fraud detection: a realistic modeling and a novel learning strategy. IEEE Transactions on Neural Networks and Learning Systems, 29(8), 3784-3797.

Content:

Data preparation

In [1]:
```
# Import modules
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
```

In [2]:

```
# Load data
df = pd.read_csv('data/creditcard.csv', header=None)
df.describe()
```

Out[2]:

	0	1	2	3	4	5
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e-15
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01

8 rows × 31 columns

In [3]:
```
# Print data types
print(df.dtypes)
print(df.columns)
```

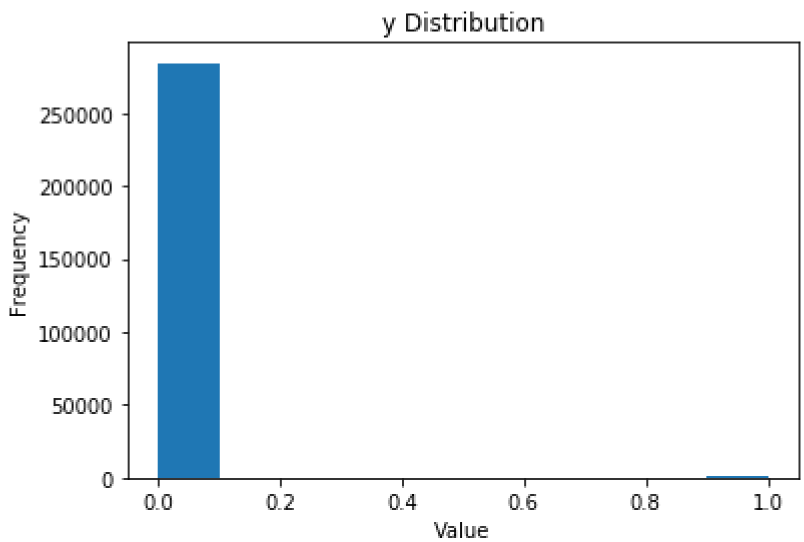
```
0     float64
1     float64
2     float64
3     float64
4     float64
5     float64
6     float64
7     float64
8     float64
9     float64
10    float64
11    float64
12    float64
13    float64
14    float64
15    float64
16    float64
17    float64
18    float64
19    float64
20    float64
21    float64
22    float64
23    float64
24    float64
25    float64
26    float64
27    float64
28    float64
29    float64
30      int64
dtype: object
Int64Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
            17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
           dtype='int64')
```

In [4]:
```
# Define X,y
X = df.iloc[:,:-2]
y = df.iloc[:,-1]
```

In [5]:
```python
# Plot histogram
yhist = plt.hist(y)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('y Distribution')

# Count transactions
pos = sum(y)
pos_rel = sum(y)/y.shape[0]
print("Number of transactions: {}".format(y.shape[0]))
print("Anomalies: {}".format(pos))
print("Anomalies percentage of all transactions: %.4f%%" % (pos_rel*100))
```

```
Number of transactions: 284807
Anomalies: 492
Anomalies percentage of all transactions: 0.1727%
```



Summary of the credit card transaction dataset:

- The variables are unnamed, because we are not working with original credit card data but with variables that have been orthogonally transformed into uncorrelated variables (principal components).
- There are 29 variables of type float and 1 variable of type integer. The former are the principal component that we can use as features, and the latter is the binary response variable indicating the transaction anomalies.

- The number of anomalies is very small compared to the total number of transactions collected over the course of two days. In other words, the dataset is highly unbalanced, and this requires special attention when we build a classifier to predict anomalies in the transaction data.

Logistic Regression

Predicting anomalies is a binary classification problem. We assume that a transaction represents either an anomaly (1) or not (0), but never both. Logistic Regression is a good starting point for this type of classification because it is fast, and still allows us to explain what variables are influential. (While this is always the case, note that our variables are PCAs meaning that their influence does not give us any information.) We will follow a step-wise approach:

1. Split the data into a training set and a testing (or hold-out) set;
2. Scale and center the data;
3. Fit an initial classifier:
   - Use default parameters;
   - Predict (non)anomalous transactions;
   - Evaluate initial classifier;
4. Hyperparameter tuning of classifier with k-fold cross-validation:
   - Identify optimized parameter set for classifier;
   - Predict (non)anomalous transactions;
   - Evaluate optimized classifier.

In [6]:
```
# Import modules
from inspect import signature
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, r
from sklearn.model_selection import cross_val_score, train_test_split, Randomize
from sklearn.preprocessing import StandardScaler
```

In [7]:
```
# Create training and testing sets
X_train, X_test, y_train, y_test  = train_test_split(X, y, test_size = 0.3, rand
```

In [8]:
```
# Scale and center data
scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

In [9]:
```
# Instantiate classifier
clf = LogisticRegression(solver='lbfgs', max_iter=200, random_state=21)

# Fit classifier to the training set
clf.fit(X_train, y_train)
```

Out[9]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=200,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=21, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [10]:
```
# Compute training metrics
accuracy = clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('Initial LR training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
===========================Initial LR training report===========================
=
Accuracy: 0.9992
MSE: 0.0008
Confusion matrix:
[[198993     28]
 [   132    211]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       0.88      0.62      0.73       343

    accuracy                           1.00    199364
   macro avg       0.94      0.81      0.86    199364
weighted avg       1.00      1.00      1.00    199364
```

In [11]:
```python
# Compute testing metrics
accuracy = clf.score(X_test, y_test)

# Predict labels of test set
y_pred = clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('Initial LR testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
===========================Initial LR testing report===========================
=
Accuracy: 0.9992
MSE: 0.0008
Confusion matrix:
[[85283    11]
 [   61    88]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.89      0.59      0.71       149

    accuracy                           1.00     85443
   macro avg       0.94      0.80      0.85     85443
weighted avg       1.00      1.00      1.00     85443
```

	Prediction: 0	Prediction: 1
Actual: 0	True negative	False positive
Actual: 1	False negative	True positive

- Precision = tp / (tp + fp)
- Recall = tp / (tp + fn)
- F-beta score = 2 * (precision * recall) / (precision + recall)

The classification report (above) shows that the accuracy of the model is outstanding (close to 1.00). But this does not mean this a good model. Why? The vast majority - 99.83% of the transactions are not marked as anomalies in the dataset. An alternative model would simply classify all values as 0 (not anomalous), and still have an accuracy of 1.00 (98.83% to be exact). Despite the fact that this classifier has a very high accuracy and a very low mean squared error, we need to evaluate the metrics that reflect the fact that this dataset is highly unbalanced. We want to focus particularly on the class of interest: anomalous transactions. The recall rate for anomalies (value 1), is not indeed very high (0.59). While the precision for anomalies (0.89) is a good result for an uncalibrated model, the confusion matrix shows that we incorrectly classified 61 transactions as anomalous.

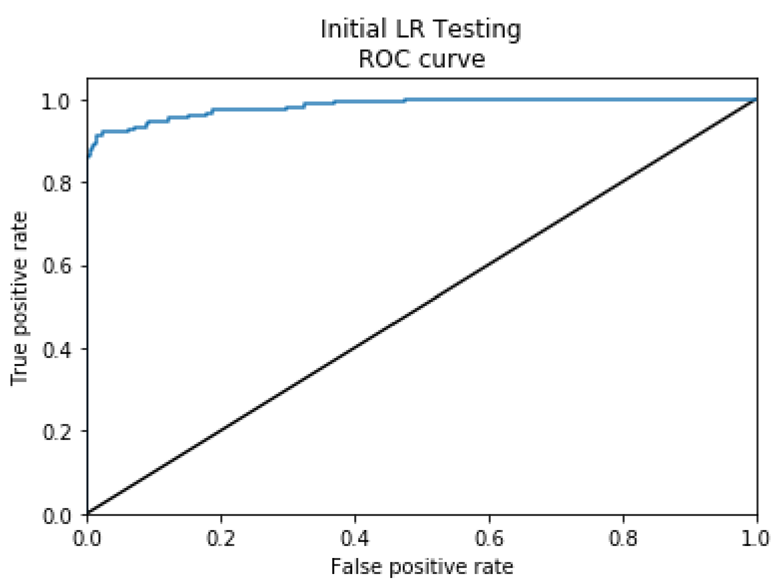
In [12]:
```
# Compute predicted probabilities
y_pred_prob = clf.predict_proba(X_test)[:,1]

# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Initial LR Testing\nROC curve')
plt.show()
```

```
AUC: 0.9829062620044716
```

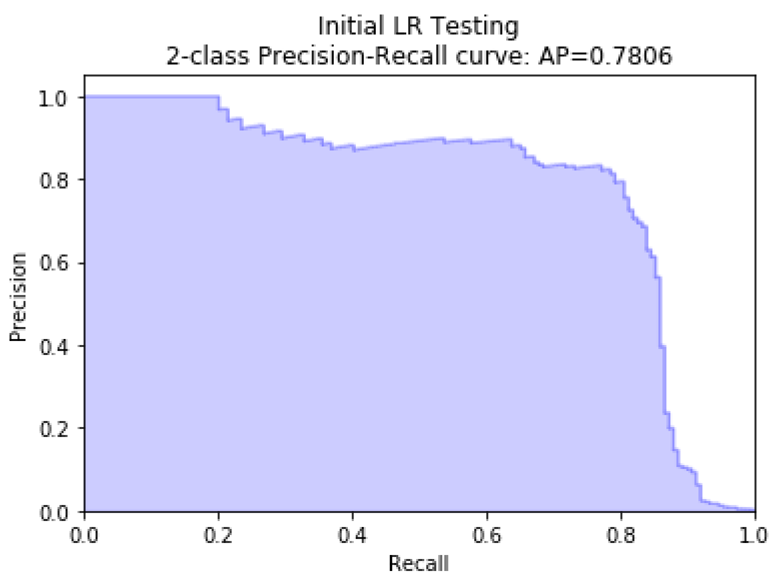


In [13]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Initial LR Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.form
plt.show()
```

```
AUPRC: 0.7805894031351092
```



Again, we see that metrics like area under ROC curve and accuracy (above) give a too optimistic impression of model performance. To reflect the unbalanced character of the credit card transaction data (0.1727% of the transactions were anomalous and 99.8273% were not anomalous), we also plotted the Precision-Recall curve. The area under the Precision-Recall curve (AUPRC) is a useful metric: if we had to assign this model a grade, 78/100 would be it. Not bad, but still much unexploited potential.

LR hyperparameter tuning

To improve the performance of the Logistic Regression classifier, we will calibrate its main parameter C with a random grid search. The C-parameter fixes the inverse of regularization strength, and setting this parameter to a smaller value will increase the regularization strength.

In [14]:
```
# Define hyperparameter grid
c_space = np.logspace(-3, 2, 51)
rand_grid = {'C': c_space,
             'solver': ['lbfgs'] }
print(rand_grid)

# Instantiate search object (use all cores but one)
grid = RandomizedSearchCV(LogisticRegression(random_state=21, max_iter=100), rand
                          n_iter = 20, cv=2, random_state=21, n_jobs = -2, verbo

# Fit object to data
grid.fit(X_train, y_train)

# Extract best model
optimized_clf = grid.best_estimator_
```

```
{'C': array([1.00000000e-03, 1.25892541e-03, 1.58489319e-03, 1.99526231e-03,
       2.51188643e-03, 3.16227766e-03, 3.98107171e-03, 5.01187234e-03,
       6.30957344e-03, 7.94328235e-03, 1.00000000e-02, 1.25892541e-02,
       1.58489319e-02, 1.99526231e-02, 2.51188643e-02, 3.16227766e-02,
       3.98107171e-02, 5.01187234e-02, 6.30957344e-02, 7.94328235e-02,
       1.00000000e-01, 1.25892541e-01, 1.58489319e-01, 1.99526231e-01,
       2.51188643e-01, 3.16227766e-01, 3.98107171e-01, 5.01187234e-01,
       6.30957344e-01, 7.94328235e-01, 1.00000000e+00, 1.25892541e+00,
       1.58489319e+00, 1.99526231e+00, 2.51188643e+00, 3.16227766e+00,
       3.98107171e+00, 5.01187234e+00, 6.30957344e+00, 7.94328235e+00,
       1.00000000e+01, 1.25892541e+01, 1.58489319e+01, 1.99526231e+01,
       2.51188643e+01, 3.16227766e+01, 3.98107171e+01, 5.01187234e+01,
       6.30957344e+01, 7.94328235e+01, 1.00000000e+02]), 'solver': ['lbfgs']}
Fitting 2 folds for each of 20 candidates, totalling 40 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  27 tasks      | elapsed:   15.4s
[Parallel(n_jobs=-2)]: Done  40 out of  40 | elapsed:   20.0s finished
```

In [15]:
```
# Print the tuned parameters and score
print('{:=^80}'.format('LR parameters for best candidate'))
print("Optimized Parameters: {}".format(grid.best_params_))
print("All Parameters: {}".format(optimized_clf.get_params()))
print("Best score is {}".format(grid.best_score_))
```

```
=======================LR parameters for best candidate=======================
=
Optimized Parameters: {'solver': 'lbfgs', 'C': 0.12589254117941676}
All Parameters: {'C': 0.12589254117941676, 'class_weight': None, 'dual': False,
'fit_intercept': True, 'intercept_scaling': 1, 'l1_ratio': None, 'max_iter': 10
0, 'multi_class': 'warn', 'n_jobs': None, 'penalty': 'l2', 'random_state': 21,
'solver': 'lbfgs', 'tol': 0.0001, 'verbose': 0, 'warm_start': False}
Best score is 0.9991723681306555
```

In [16]:
```
# Compute training metrics
accuracy = optimized_clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = optimized_clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('Optimized LR training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
=========================Optimized LR training report=========================
=
Accuracy: 0.9992
MSE: 0.0008
Confusion matrix:
[[198993     28]
 [   133    210]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       0.88      0.61      0.72       343

    accuracy                           1.00    199364
   macro avg       0.94      0.81      0.86    199364
weighted avg       1.00      1.00      1.00    199364
```

In [17]:

```python
# Compute testing metrics
accuracy = optimized_clf.score(X_test, y_test)

# Predict labels of test set
y_pred = optimized_clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('Optimized LR testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
==========================Optimized LR testing report==========================
=
Accuracy: 0.9991
MSE: 0.0009
Confusion matrix:
[[85283    11]
 [   62    87]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.89      0.58      0.70       149

    accuracy                           1.00     85443
   macro avg       0.94      0.79      0.85     85443
weighted avg       1.00      1.00      1.00     85443
```

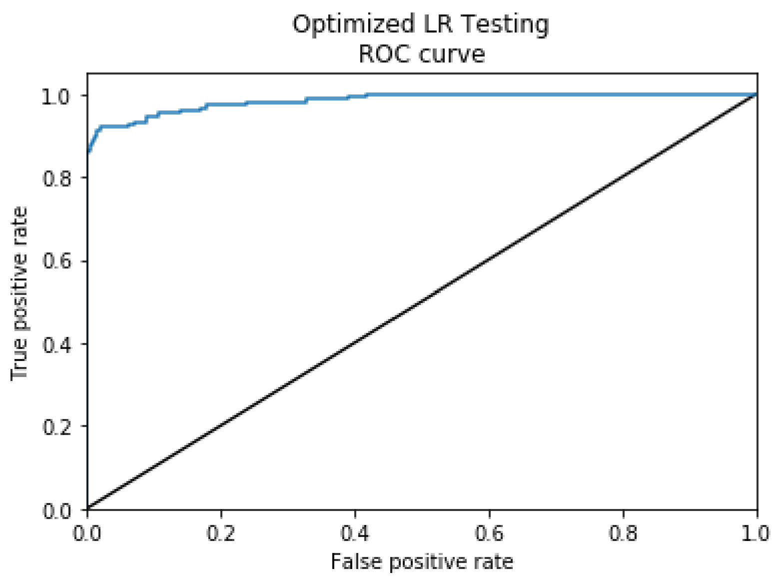
In [18]:
```
# Compute predicted probabilities
y_pred_prob = optimized_clf.predict_proba(X_test)[:,1]

# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Optimized LR Testing\nROC curve')
plt.show()
```

```
AUC: 0.9839071428110556
```

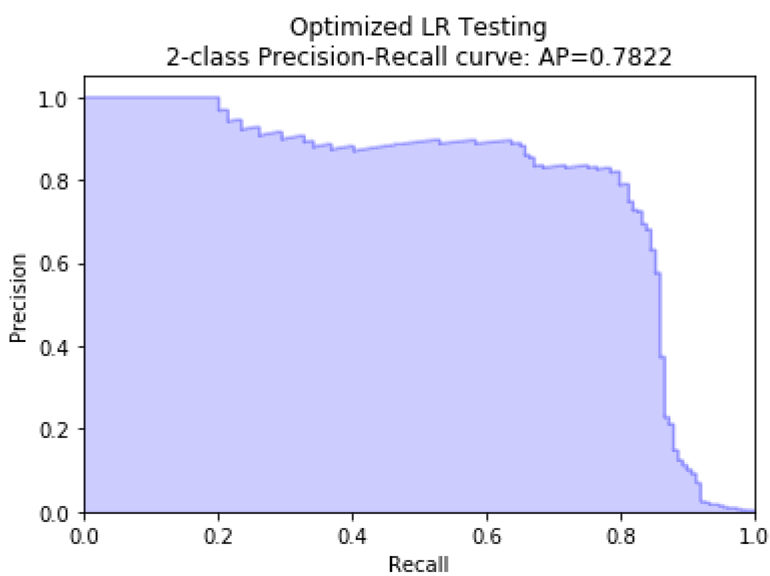


In [19]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Optimized LR Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.fo
plt.show()
```

```
AUPRC: 0.7821883145333293
```



Logistic Regression final performance (above): The model performance indicated by the AUPRC metric (area under Precision-Recall curve) did not improve substantially in comparison to the initial model, so it is time to try a different classifier.

Kernel Support Vector Machine classification (SVC)

Next, we will fit a kernel-type support vector classifier (SVC) with Gaussian radial basis function (RBF). Where logistic regression uses the output of a linear model, the SVC will define a hyperplane within the N-dimensional parameter space to classify the data points into either of the two categories--1 for anomalous transactions and 0 for all other transactions. This parameter space consists of the set of N predictor or feature variables. We will follow the same step-wise approach as for LR, starting with an initial model followed by hyperparameter tuning.

In [20]:
```python
# Import modules
from sklearn.svm import SVC
```

In [21]:
```python
# Instantiate classifier (turning off probability increases speed)
clf = SVC(probability=True, gamma='scale', max_iter=-1, random_state=21)

# Fit classifier to training set
clf.fit(X_train, y_train)
```

Out[21]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=True, random_state=21, shrinking=True, tol=0.001,
    verbose=False)
```

In [22]:
```python
# Compute training metrics
accuracy = clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('Initial SVC training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
==========================Initial SVC training report==========================
=
Accuracy: 0.9997
MSE: 0.0003
Confusion matrix:
[[199015      6]
 [    56    287]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       0.98      0.84      0.90       343

    accuracy                           1.00    199364
   macro avg       0.99      0.92      0.95    199364
weighted avg       1.00      1.00      1.00    199364
```

In [23]:
```python
# Compute testing metrics
accuracy = clf.score(X_test, y_test)

# Predict labels of test set
y_pred = clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('Initial SVC testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
===========================Initial SVC testing report===========================
=
Accuracy: 0.9993
MSE: 0.0007
Confusion matrix:
[[85288     6]
 [   51    98]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.94      0.66      0.77       149

    accuracy                           1.00     85443
   macro avg       0.97      0.83      0.89     85443
weighted avg       1.00      1.00      1.00     85443
```

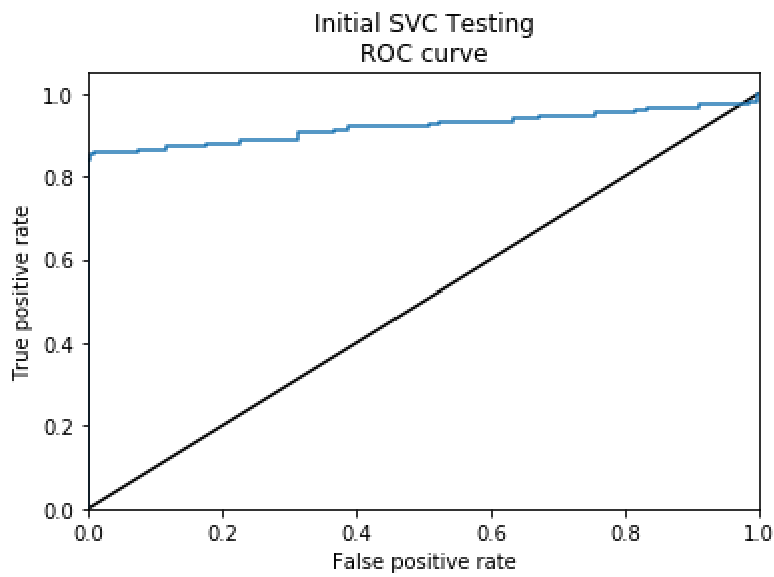
In [24]:
```
# Compute predicted probabilities
y_pred_prob = clf.predict_proba(X_test)[:,1]

# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Initial SVC Testing\nROC curve')
plt.show()
```

```
AUC: 0.9198611576886137
```

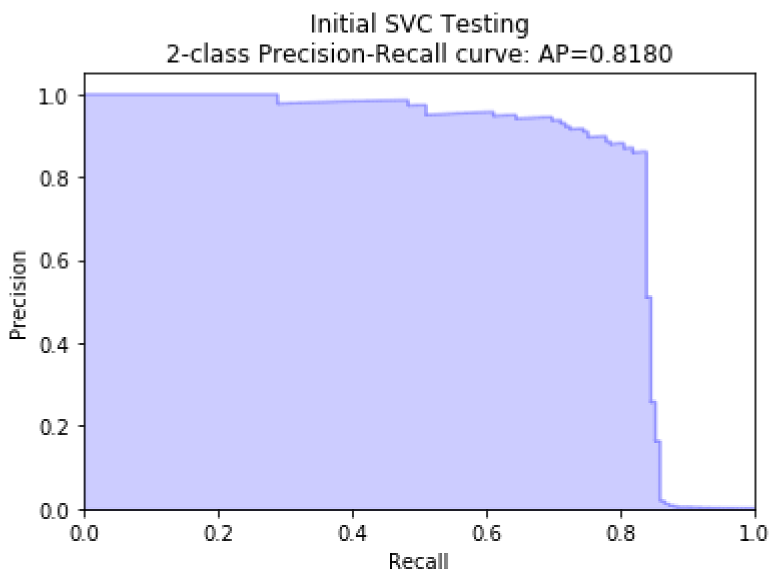


In [25]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Initial SVC Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.for
plt.show()
```

```
AUPRC: 0.8179754128130251
```



The Kernel SVC (above) performs better than LR because it predicts more true positives (anomalies recall=66%) and less false positives (anomalies precision=94%). At 81%, the SVC AUPRC, main metric of interest, is also greater than for LR.

Kernel SVC hyperparameter tuning

To optimize the Kernel SVC, we will tune parameters C and gamma. Gamma defines the nonlinear hyperplane of the SVC, and represents the inverse of the radius of influence of samples identified by the model as support vectors. Because gamma determines how closely the hyperplane fits the training set, it follows that high values of gamma can lead to overfitting. Therefore, we use a moderate range. Additionally, we limit the number of iterations in the event that the classifier does not converge toward a solution.

In [26]:
```
# Define hyperparameter grid
c_space = np.logspace(-3, 2, 51)
gamma_space = np.logspace(-3, 2, 51)
rand_grid = {'C': c_space,
             'gamma': gamma_space}
print(rand_grid)
```

```
{'C': array([1.00000000e-03, 1.25892541e-03, 1.58489319e-03, 1.99526231e-03,
       2.51188643e-03, 3.16227766e-03, 3.98107171e-03, 5.01187234e-03,
       6.30957344e-03, 7.94328235e-03, 1.00000000e-02, 1.25892541e-02,
       1.58489319e-02, 1.99526231e-02, 2.51188643e-02, 3.16227766e-02,
       3.98107171e-02, 5.01187234e-02, 6.30957344e-02, 7.94328235e-02,
       1.00000000e-01, 1.25892541e-01, 1.58489319e-01, 1.99526231e-01,
       2.51188643e-01, 3.16227766e-01, 3.98107171e-01, 5.01187234e-01,
       6.30957344e-01, 7.94328235e-01, 1.00000000e+00, 1.25892541e+00,
       1.58489319e+00, 1.99526231e+00, 2.51188643e+00, 3.16227766e+00,
       3.98107171e+00, 5.01187234e+00, 6.30957344e+00, 7.94328235e+00,
       1.00000000e+01, 1.25892541e+01, 1.58489319e+01, 1.99526231e+01,
       2.51188643e+01, 3.16227766e+01, 3.98107171e+01, 5.01187234e+01,
       6.30957344e+01, 7.94328235e+01, 1.00000000e+02]), 'gamma': array([1.0000
0000e-03, 1.25892541e-03, 1.58489319e-03, 1.99526231e-03,
       2.51188643e-03, 3.16227766e-03, 3.98107171e-03, 5.01187234e-03,
       6.30957344e-03, 7.94328235e-03, 1.00000000e-02, 1.25892541e-02,
       1.58489319e-02, 1.99526231e-02, 2.51188643e-02, 3.16227766e-02,
       3.98107171e-02, 5.01187234e-02, 6.30957344e-02, 7.94328235e-02,
       1.00000000e-01, 1.25892541e-01, 1.58489319e-01, 1.99526231e-01,
       2.51188643e-01, 3.16227766e-01, 3.98107171e-01, 5.01187234e-01,
       6.30957344e-01, 7.94328235e-01, 1.00000000e+00, 1.25892541e+00,
       1.58489319e+00, 1.99526231e+00, 2.51188643e+00, 3.16227766e+00,
       3.98107171e+00, 5.01187234e+00, 6.30957344e+00, 7.94328235e+00,
       1.00000000e+01, 1.25892541e+01, 1.58489319e+01, 1.99526231e+01,
       2.51188643e+01, 3.16227766e+01, 3.98107171e+01, 5.01187234e+01,
       6.30957344e+01, 7.94328235e+01, 1.00000000e+02])}
```

In [27]:
```
# Instantiate RandomizedSearchCV object (use all cores but one)
grid = RandomizedSearchCV(SVC(probability=True, random_state=21, max_iter = 1000
                          n_iter = 20, cv=2, random_state=21, n_jobs = -2, verbo

# Fit object to data
grid.fit(X_train, y_train)

# Extract best model
optimized_clf = grid.best_estimator_
```

```
Fitting 2 folds for each of 20 candidates, totalling 40 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  27 tasks      | elapsed: 12.2min
C:\Users\Dennis\Anaconda3\lib\site-packages\joblib\externals\loky\process_execu
tor.py:706: UserWarning: A worker stopped while some jobs were given to the exe
cutor. This can be caused by a too short worker timeout or by a memory leak.
  "timeout or by a memory leak.", UserWarning
[Parallel(n_jobs=-2)]: Done  40 out of  40 | elapsed: 16.1min finished
C:\Users\Dennis\Anaconda3\lib\site-packages\sklearn\svm\base.py:241: Convergenc
eWarning: Solver terminated early (max_iter=1000).  Consider pre-processing you
r data with StandardScaler or MinMaxScaler.
  % self.max_iter, ConvergenceWarning)
```

In [28]:
```
# Print the tuned parameters and score
print('{:=^80}'.format('SVC parameters for best candidate'))
print("Optimized Parameters: {}".format(grid.best_params_))
print("All Parameters: {}".format(optimized_clf.get_params()))
print("Best score is {}".format(grid.best_score_))
```

```
=======================SVC parameters for best candidate=======================
=
Optimized Parameters: {'gamma': 0.003981071705534973, 'C': 39.81071705534978}
All Parameters: {'C': 39.81071705534978, 'cache_size': 200, 'class_weight': Non
e, 'coef0': 0.0, 'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 0.0039
81071705534973, 'kernel': 'rbf', 'max_iter': 1000, 'probability': True, 'random
_state': 21, 'shrinking': True, 'tol': 0.001, 'verbose': False}
Best score is 0.9994783411247767
```

In [29]:
```python
# Compute training metrics
accuracy = optimized_clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = optimized_clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('Optimized SVC training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
=========================Optimized SVC training report=========================
=
Accuracy: 0.9997
MSE: 0.0003
Confusion matrix:
[[199016      5]
 [    55    288]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       0.98      0.84      0.91       343

    accuracy                           1.00    199364
   macro avg       0.99      0.92      0.95    199364
weighted avg       1.00      1.00      1.00    199364
```

In [30]:
```
# Compute testing metrics
accuracy = optimized_clf.score(X_test, y_test)

# Predict labels of test set
y_pred = optimized_clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('Optimized SVC testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
==========================Optimized SVC testing report==========================
=
Accuracy: 0.9995
MSE: 0.0005
Confusion matrix:
[[85286     8]
 [   37   112]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.93      0.75      0.83       149

    accuracy                           1.00     85443
   macro avg       0.97      0.88      0.92     85443
weighted avg       1.00      1.00      1.00     85443
```

In [31]:

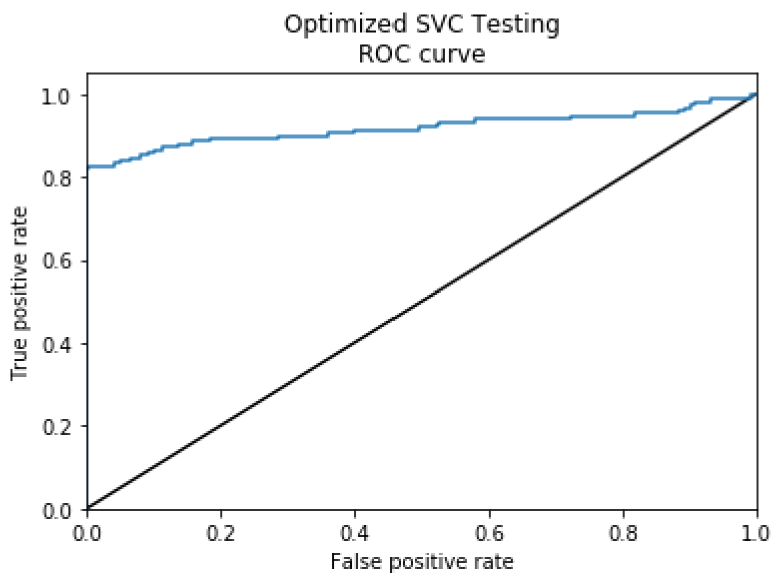
```
# Compute predicted probabilities
y_pred_prob = optimized_clf.predict_proba(X_test)[:,1]

# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Optimized SVC Testing\nROC curve')
plt.show()
```

```
AUC: 0.9177951886274762
```

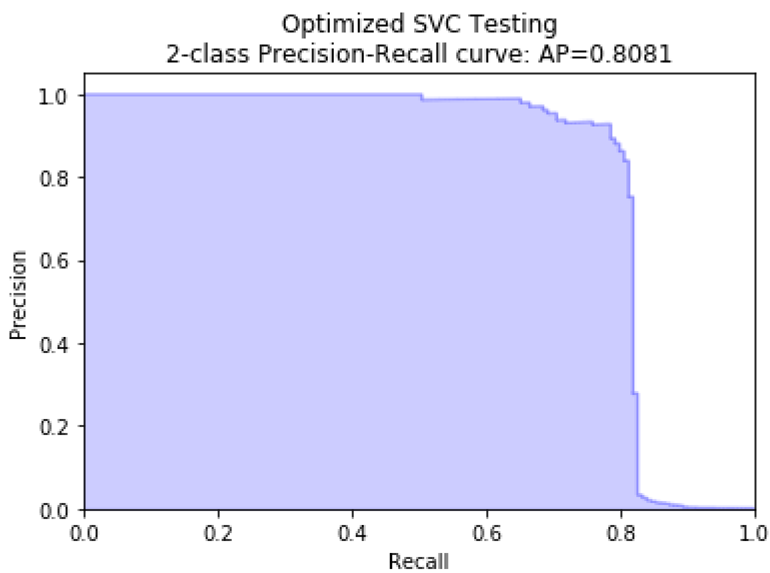


In [32]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Optimized SVC Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.fo
plt.show()
```

```
AUPRC: 0.8080783333481538
```



Kernel Support Vector Classifier final performance (above):

- Hyperparameter tuning the Kernel SVC did not result in a better classifier than the Kernel SVC we started with.
- Some Kernel SVC parameter combination failed to converge within the maximum number of iterations specified.
- Regardless, Kernel SVC performed notably better than the LR in terms of precision (low number of false negatives and false positives), recall and area under Precision-Recall curve (AUPRC).

At this point we could decide to try again and explore the parameter space in more detail, but let's try other classifiers instead.

Gradient Boosting Model (GBM) classification

Gradient Boosting builds an additive model in a forward step-wise approach, by fitting a single regression tree (in binary classification) that optimizes the deviance loss function. As such, a GBM combines both parametric and non-parametric methods.

In [33]:
```
# Import modules
from sklearn.ensemble import GradientBoostingClassifier
```

In [34]:
```
# Instantiate classifier
clf = GradientBoostingClassifier(random_state=21, verbose=1)

# Fit classifier to the training set
clf.fit(X_train, y_train)
```

```
      Iter       Train Loss   Remaining Time 
         1           0.0237            1.49m
         2 24424088907.7394            1.63m
         3 24424088907.7335            1.56m
         4 24424088442.4370            1.52m
         5 24424088442.4367            1.46m
         6 24424088442.4364            1.41m
         7 24424088442.4363            1.37m
         8 24424088442.4362            1.33m
         9 24424088442.4361            1.30m
        10 24424088442.4360            1.28m
        20 24424088442.4355            1.10m
        30 5612946226909993688157388 8.0000           56.52s
        40 5612946226909993688157388 8.0000           47.88s
        50 5612946226909993688157388 8.0000           39.37s
        60 5612946226909993688157388 8.0000           31.32s
        70 5612946226909993688157388 8.0000           23.45s
        80 5612946226909993688157388 8.0000           15.63s
        90 5612946226909993688157388 8.0000            7.78s
       100 5612946226909993688157388 8.0000            0.00s
```

Out[34]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
                           n_iter_no_change=None, presort='auto',
                           random_state=21, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=1,
                           warm_start=False)
```

In [35]:
```
# Compute training metrics
accuracy = clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('Initial SVC training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
==========================Initial SVC training report===========================
=
Accuracy: 0.9992
MSE: 0.0008
Confusion matrix:
[[198998     23]
 [   132    211]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       0.90      0.62      0.73       343

    accuracy                           1.00    199364
   macro avg       0.95      0.81      0.87    199364
weighted avg       1.00      1.00      1.00    199364
```

In [36]:
```python
# Compute testing metrics
accuracy = clf.score(X_test, y_test)

# Predict labels of test set
y_pred = clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('Initial GBM testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
===========================Initial GBM testing report===========================
=
Accuracy: 0.9993
MSE: 0.0007
Confusion matrix:
[[85284    10]
 [   51    98]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.91      0.66      0.76       149

    accuracy                           1.00     85443
   macro avg       0.95      0.83      0.88     85443
weighted avg       1.00      1.00      1.00     85443
```

In [37]:

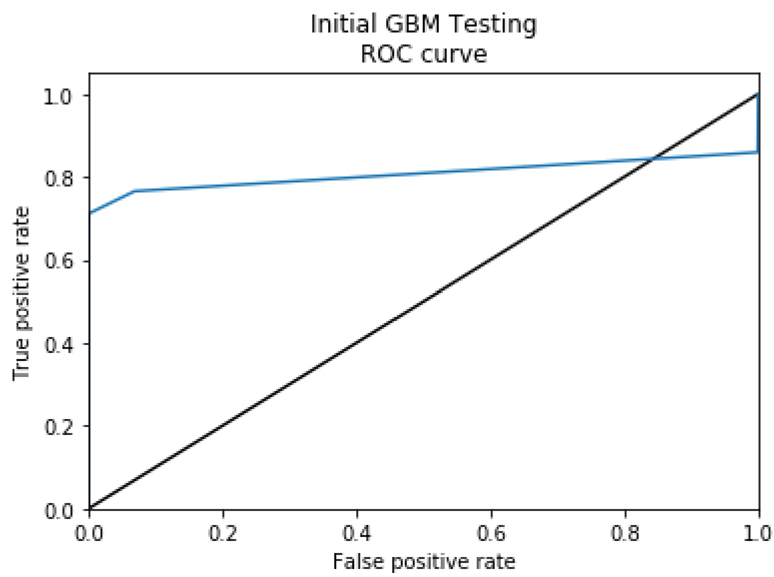
```
# Compute predicted probabilities
y_pred_prob = clf.predict_proba(X_test)[:,1]

# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Initial GBM Testing\nROC curve')
plt.show()
```

```
AUC: 0.8070295116630154
```

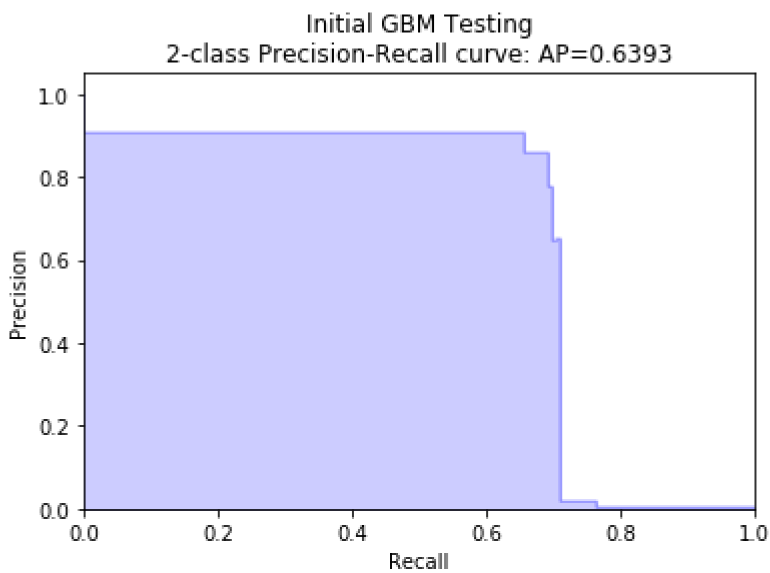


In [38]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Initial GBM Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.fori
plt.show()
```

```
AUPRC: 0.6392519105493971
```



GBM hyperparameter tuning

In [39]:
```
# Define hyperparameter grid
learning_rate = [0.02, 0.1]
n_estimators = [int(x) for x in [100, 200, 300, 400]]
subsample = [0.5, 0.9]
max_depth = [int(x) for x in [3, 4, 5, 10]]
min_samples_split = [int(x) for x in [2, 3, 4, 5]]
rand_grid = {'learning_rate': learning_rate,
             'n_estimators': n_estimators,
             'subsample': subsample,
             'max_depth': max_depth,
             'min_samples_split': min_samples_split}
print(rand_grid)
```

```
{'learning_rate': [0.02, 0.1], 'n_estimators': [100, 200, 300, 400], 'subsampl
e': [0.5, 0.9], 'max_depth': [3, 4, 5, 10], 'min_samples_split': [2, 3, 4, 5]}
```

In [40]:
```
# Instantiate RandomizedSearchCV object (use all cores but one)
grid = RandomizedSearchCV(GradientBoostingClassifier(validation_fraction=0.3, n_
                          n_iter = 20, cv=2, random_state=21, n_jobs = -2, verbo

# Fit object to data
grid.fit(X_train, y_train)

# Extract best model
optimized_clf = grid.best_estimator_
```

```
Fitting 2 folds for each of 20 candidates, totalling 40 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  27 tasks      | elapsed:  2.9min
[Parallel(n_jobs=-2)]: Done  40 out of  40 | elapsed:  4.2min finished
```

In [41]:
```
# Print the tuned parameters and score
print('{:=^80}'.format('GBM parameters for best candidate'))
print("Optimized Parameters: {}".format(grid.best_params_))
print("All Parameters: {}".format(optimized_clf.get_params()))
print("Best score is {}".format(grid.best_score_))
```

```
=======================GBM parameters for best candidate=======================
=
Optimized Parameters: {'subsample': 0.9, 'n_estimators': 300, 'min_samples_spli
t': 2, 'max_depth': 4, 'learning_rate': 0.02}
All Parameters: {'criterion': 'friedman_mse', 'init': None, 'learning_rate': 0.
02, 'loss': 'deviance', 'max_depth': 4, 'max_features': None, 'max_leaf_nodes':
None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_le
af': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimator
s': 300, 'n_iter_no_change': 10, 'presort': 'auto', 'random_state': 21, 'subsam
ple': 0.9, 'tol': 0.0001, 'validation_fraction': 0.3, 'verbose': 0, 'warm_star
t': False}
Best score is 0.9992626552436749
```

In [42]:
```python
# Compute training metrics
accuracy = optimized_clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = optimized_clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('Optimized GBM training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
=========================Optimized GBM training report=========================
=
Accuracy: 0.9995
MSE: 0.0005
Confusion matrix:
[[198997     24]
 [    67    276]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       0.92      0.80      0.86       343

    accuracy                           1.00    199364
   macro avg       0.96      0.90      0.93    199364
weighted avg       1.00      1.00      1.00    199364
```

In [43]:
```python
# Compute testing metrics
accuracy = optimized_clf.score(X_test, y_test)

# Predict labels of test set
y_pred = optimized_clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('Optimized GBM testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
==========================Optimized GBM testing report==========================
=
Accuracy: 0.9994
MSE: 0.0006
Confusion matrix:
[[85280    14]
 [   41   108]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.89      0.72      0.80       149

    accuracy                           1.00     85443
   macro avg       0.94      0.86      0.90     85443
weighted avg       1.00      1.00      1.00     85443
```

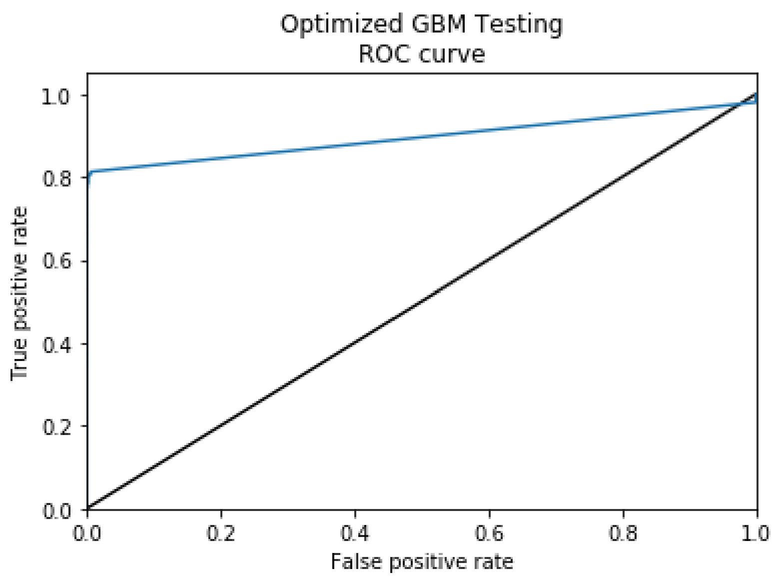
In [44]:
```
# Compute predicted probabilities
y_pred_prob = optimized_clf.predict_proba(X_test)[:,1]

# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Optimized GBM Testing\nROC curve')
plt.show()
```

```
AUC: 0.89534174178125
```

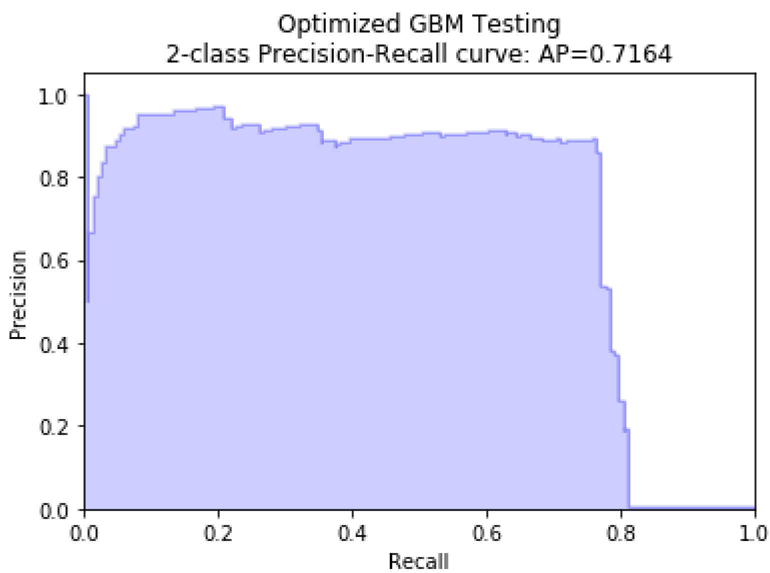


```
In [45]: # Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Optimized GBM Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.f
plt.show()
```

```
AUPRC: 0.716365268143389
```



With an AUPRC of 0.7164 (above), the gradient boosting model performed worse than both the LR classifier and Kernel SVC. Lower performance is mostly explained by the lower recall for anomalies (value 1). While the GBM was on point for the anomalies it identified, it also missed a large percentage of anomalies--at least more than the LR classifier and Kernel SVC.

Random Forest (RF) classification

Until now we used LR, Kernel SVC and GBM to make predictions, and all three have limitations when it comes to classification of unbalanced data. LR is a special case of generalized linear model (GLM) and makes assumptions about the underlying data distribution. This method works best with uncorrelated data and logarithmic error distributions, and therefore requires many data samples for fitting. Kernel SVC fitting involves the computation of polynomial surfaces, and the nonlinear nature of polynomial calculations makes that Kernel SVCs are not easily parallelized.

GBM builds trees one at a time, each attempting to explain the residual error of the previous tree. While this can work for balanced datasets, the sequential nature of this process can be a disadvantage.

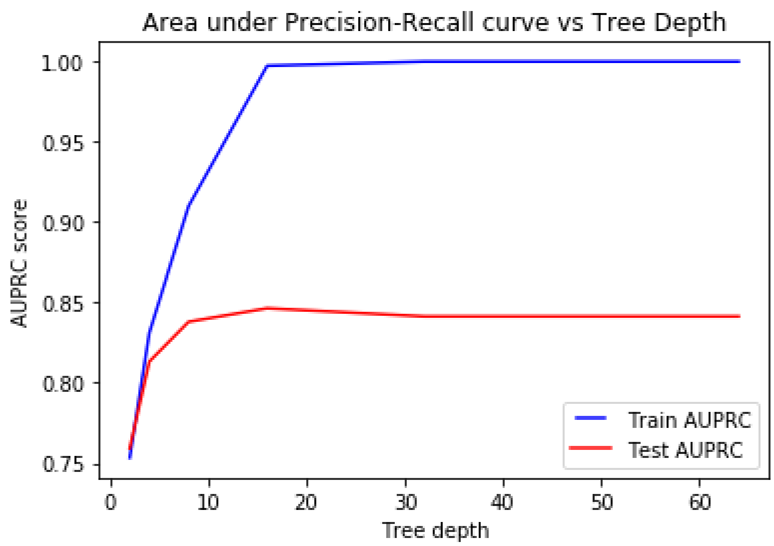
Random Forest (RF) classification offers a non-parametric approach where a large number of uncorrelated models (decision trees) are fitted to the data, and vote independently as a joint committee on what the outcome of each prediction should be. One advantage of RF is that it makes no assumptions about the sample distribution or error distribution, meaning the classifier is robust and not biased by outliers. Furthermore, RF can be parallelized into as many processes as there are estimators (trees). Because a RF calculates fast, we will not hypertune the parameters but instead allow it to fit the data without constraints.

In [46]:
```
# Import modules
from sklearn.ensemble import RandomForestClassifier
```

In [47]:
```
# Fit random forest for range of maximum depth of tree
max_depths = [int(x) for x in [2,4,8,16,32,64]]

train_results = []
test_results = []
for max_depth in max_depths:
   clf = RandomForestClassifier(max_depth=max_depth, n_estimators=100, random_st
   clf.fit(X_train, y_train)
   train_pred_prob = clf.predict_proba(X_train)[:,1]
   average_precision = average_precision_score(y_train, train_pred_prob)
   train_results.append(average_precision)
   y_pred_prob = clf.predict_proba(X_test)[:,1]
   average_precision = average_precision_score(y_test, y_pred_prob)
   test_results.append(average_precision)
```

In [48]:
```
# Plot AUPRC vs tree depth
from matplotlib.legend_handler import HandlerLine2D
line1, = plt.plot(max_depths, train_results, 'b', label="Train AUPRC")
line2, = plt.plot(max_depths, test_results, 'r', label="Test AUPRC")
plt.legend(handler_map={line1: HandlerLine2D(numpoints=2)})
plt.ylabel('AUPRC score')
plt.xlabel('Tree depth')
plt.title('Area under Precision-Recall curve vs Tree Depth')
plt.show()
```



The maximum area under the Precision-Recall curve (AUPRC) is reached for a tree depth less than 20. It appears that an AUPRC of ~0.84 is the best possible testing performance we may expect given a training AUPRC of close to 1.00.

Random Forest without constraints

In [49]:
```
# Import modules
from sklearn.ensemble import RandomForestClassifier
```

In [50]:
```
# Zero ratios
y_train_s = np.prod(y_train.shape)
y_train_z = (y_train_s - np.sum(y_train)) / y_train_s
y_test_s = np.prod(y_test.shape)
y_test_z = (y_test_s - np.sum(y_test)) / y_test_s
print("Zero ratio in training labels: {}".format(y_train_z))
print("Zero ratio in testing labels: {}".format(y_test_z))
```

```
Zero ratio in training labels: 0.9982795289019081
Zero ratio in testing labels: 0.9982561473731025
```

In [51]:
```python
# Compute sample weights for unbalanced classes as inverse of probability
weight_0 = 1.0
weight_1 = (1 - y_train_z)**-1
sample_weight = np.array([weight_1 if i == 1 else weight_0 for i in enumerate(y_
print("Sample weight for logical(0): {}".format(weight_0))
print("Sample weight for logical(1): {}".format(weight_1))
```

```
Sample weight for logical(0): 1.0
Sample weight for logical(1): 581.2361516035012
```

In [52]:
```python
# Instantiate classifier
clf = RandomForestClassifier(n_estimators=200, random_state=21, n_jobs = -2, verb

# Fit classifier to training set
clf = clf.fit(X_train, y_train, sample_weight=sample_weight)
```

```
building tree 183 of 200
building tree 184 of 200
building tree 185 of 200
building tree 186 of 200
building tree 187 of 200
building tree 188 of 200
building tree 189 of 200
building tree 190 of 200
building tree 191 of 200
building tree 192 of 200
building tree 193 of 200
building tree 194 of 200
building tree 195 of 200
building tree 196 of 200
building tree 197 of 200
building tree 198 of 200
building tree 199 of 200
building tree 200 of 200
```

In [53]:
```python
# Compute training metrics
accuracy = clf.score(X_train, y_train)

#  Predict labels of test set
train_pred = clf.predict(X_train)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('RF training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.2s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    1.0s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    1.4s finished
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.1s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    1.0s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    1.4s finished

===============================RF training report==============================
=
Accuracy: 1.0000
MSE: 0.0000
Confusion matrix:
[[199021      0]
 [     0    343]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       1.00      1.00      1.00       343

    accuracy                           1.00    199364
   macro avg       1.00      1.00      1.00    199364
weighted avg       1.00      1.00      1.00    199364
```

In [54]:
```
# Compute testing metrics
accuracy = clf.score(X_test, y_test)

# Predict labels of test set
y_pred = clf.predict(X_test)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('RF testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.0s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    0.3s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    0.5s finished
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.0s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    0.3s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    0.5s finished

===============================RF testing report================================
=
Accuracy: 0.9995
MSE: 0.0005
Confusion matrix:
[[85288     6]
 [   36   113]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.95      0.76      0.84       149

    accuracy                           1.00     85443
   macro avg       0.97      0.88      0.92     85443
weighted avg       1.00      1.00      1.00     85443
```

In [55]:
```
# Compute predicted probabilities
y_pred_prob = clf.predict_proba(X_test)[:,1]
```

```
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.0s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    0.4s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    0.6s finished
```

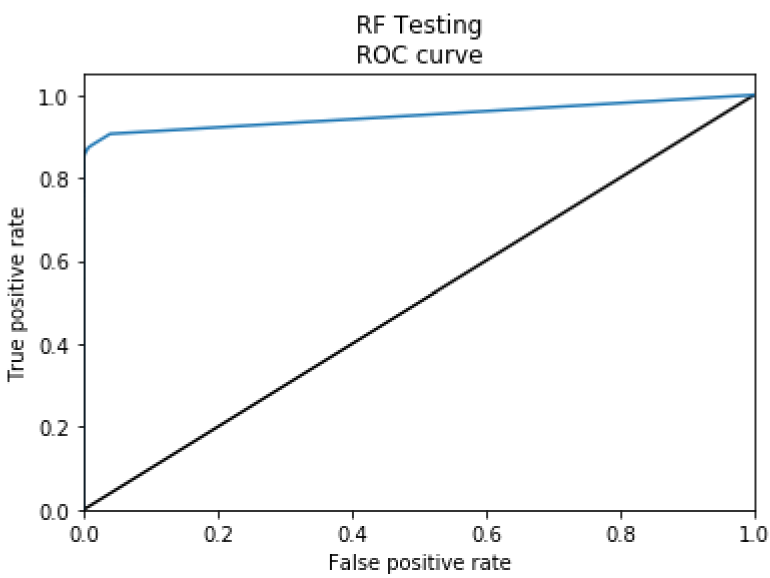
In [56]:

```
# Calculate receiver operating characteristics (ROC)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Compute AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k-')
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('RF Testing\nROC curve')
plt.show()
```

```
AUC: 0.950323814841457
```

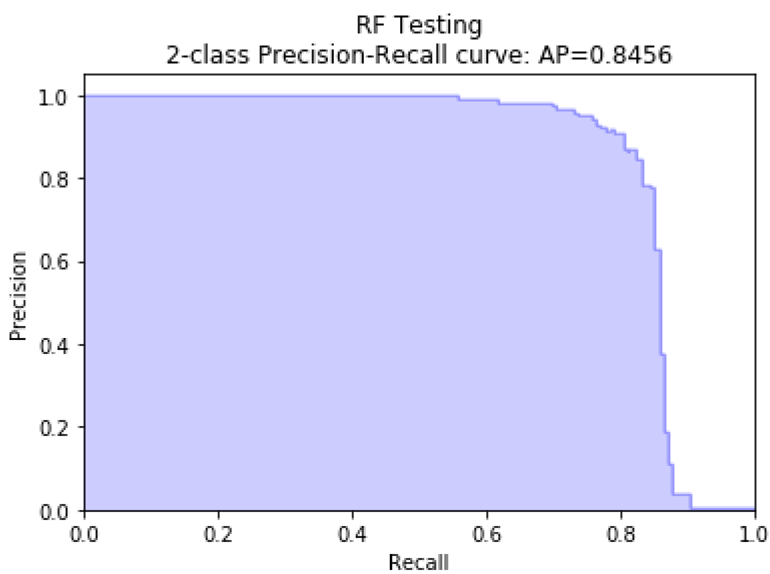


In [57]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('RF Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.format(avera
plt.show()
```

```
AUPRC: 0.8456257031199099
```



Model selection and cost-effective optimization

Selecting the best classifier: We have tested four classifiers: Logistic Regression, Kernel Support Vector Classifier, Stochastic Gradient Boosting and Random Forest. Which one is the best choice? A practical way to evaluate the suitability of a classifier for daily use is in terms of cost-effectiveness: we want the classifier to be able to make predictions on new data (high testing precision and recall), and at a low cost (fast computation and modest data requirements).

The Random Forest (AUPRC=0.8456) performed best in terms of precision and recall, followed by Kernel Support Vector Classifier (AUPRC=0.8081), Logistic Regression (AUPRC=0.7822) and Stochastic Gradient Boosting Machine (AUPRC=0.7164). RF and LR computed fastest. Conversely, Kernel SVC used the most computer time. While in a real-world scenario we might be able to obtain a better model (probably SGB or RF) given enough training data and more iterations, we assume that RF provides the most cost-effective prediction of transaction anomalies for now.

Feature importance

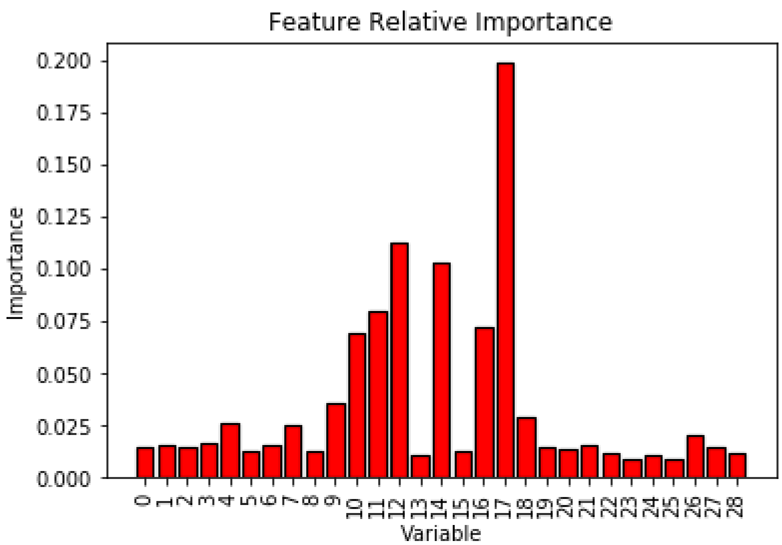
Now that we have identified the RF as the most suitable model to predict transaction anomalies, the next step is to see if can improve its cost-effectiveness by reducing data requirements. We trained the model on 29 features, so let's find out which of those features contribute the most information to the prediction.

In [58]:
```
# Get feature importances
feature_list = list(df.columns[:-1])
importances = list(clf.feature_importances_)
feature_importances = [(feature, round(importance, 4)) for feature, importance in
feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse =

# Print feature ranking
print("Feature ranking:")
_ = [print('Variable: {:3} Importance: {}'.format(*pair)) for pair in feature_im
```

```
Feature ranking:
Variable:  17 Importance: 0.1987
Variable:  12 Importance: 0.1125
Variable:  14 Importance: 0.1028
Variable:  11 Importance: 0.0793
Variable:  16 Importance: 0.0717
Variable:  10 Importance: 0.069
Variable:   9 Importance: 0.0351
Variable:  18 Importance: 0.0286
Variable:   4 Importance: 0.0259
Variable:   7 Importance: 0.0252
Variable:  26 Importance: 0.0202
Variable:   3 Importance: 0.0164
Variable:  21 Importance: 0.0155
Variable:   1 Importance: 0.0153
Variable:   6 Importance: 0.0152
Variable:  27 Importance: 0.0145
Variable:   2 Importance: 0.0142
Variable:  19 Importance: 0.0142
Variable:   0 Importance: 0.0141
Variable:  20 Importance: 0.0132
Variable:   5 Importance: 0.0123
Variable:  15 Importance: 0.0123
Variable:   8 Importance: 0.0122
Variable:  22 Importance: 0.0117
Variable:  28 Importance: 0.0113
Variable:  13 Importance: 0.011
Variable:  24 Importance: 0.0105
Variable:  25 Importance: 0.0088
Variable:  23 Importance: 0.0082
```

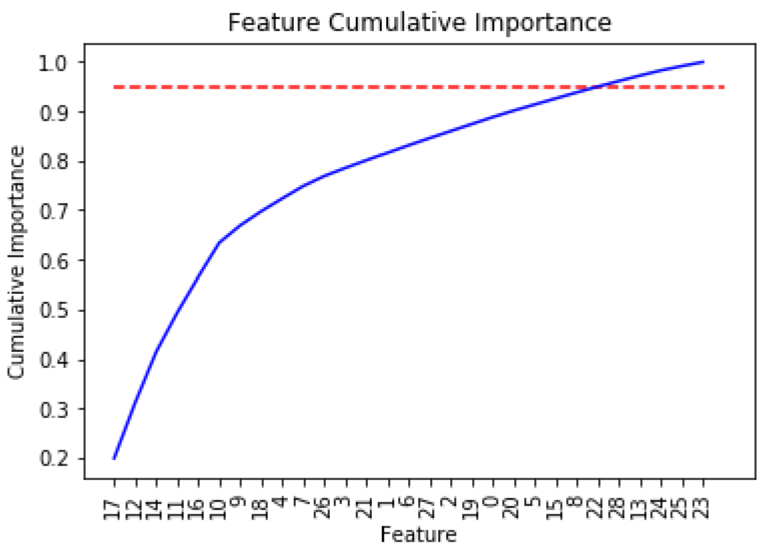
In [59]:
```
# Plot feature ranking in bar chart
X_values = list(range(len(importances)))
plt.bar(X_values, importances, orientation = 'vertical', color = 'r', edgecolor =
plt.xticks(X_values, feature_list, rotation = 'vertical')
plt.ylabel('Importance')
plt.xlabel('Variable')
plt.title('Feature Relative Importance')
plt.show()
```



In [60]:
```
# List of features sorted by decreasing importance
sorted_importances = [importance[1] for importance in feature_importances]
sorted_features = [importance[0] for importance in feature_importances]

# Cumulative importance
cumulative_importances = np.cumsum(sorted_importances)

# Create line plot
plt.plot(X_values, cumulative_importances, 'b-')
plt.hlines(y = 0.95, xmin=0, xmax=len(sorted_importances), color = 'r', linestyl
plt.xticks(X_values, sorted_features, rotation = 'vertical')
plt.xlabel('Feature')
plt.ylabel('Cumulative Importance')
plt.title('Feature Cumulative Importance')
plt.show()
```



In [61]:
```
# Number of features explaining 95% cum. importance
n_import = np.where(cumulative_importances > 0.95)[0][0] + 1
print('Number of features required (95% importance):', n_import)

# Least important features
limp_feature_names = sorted_features[-(len(importances)-n_import):]
print('Least important features (5% importance):', limp_feature_names)
```

```
Number of features required (95% importance): 24
Least important features (5% importance): [28, 13, 24, 25, 23]
```

Feature importance analysis (above) shows that 24 features out of the available set of 29 features account for 95% of the Gini Importance. There are three things we learn from this:

1. There is a relative lack of correlation between features, confirming that these are indeed orthogonally transformed data (in this case the outcome of principal component analysis);
2. The RF classifier makes optimum use of the training data;
3. We can drop 5 of the features without incurring a high cost to model performance, namely the PCAs labeled as 28, 13, 24, 25 and 23.

Retrain classifier on the most important features

In [62]:
```
# Extract the names of most important features
important_feature_names = [feature[0] for feature in feature_importances[0:(n_im

# Find the columns of the most important features
important_indices = [feature_list.index(feature) for feature in important_featur

# Create training and testing sets with only important features
X_train_imp = X_train[:,important_indices]
X_test_imp = X_test[:,important_indices]

# Print dimensions
print("Dimensions of X_train_imp: {}".format(X_train_imp.shape))
print("Dimensions of y_train_imp: {}".format(y_train.shape))
print("Dimensions of X_test_imp: {}".format(X_test_imp.shape))
print("Dimensions of y_test_imp: {}".format(y_test.shape))
```

```
Dimensions of X_train_imp: (199364, 23)
Dimensions of y_train_imp: (199364,)
Dimensions of X_test_imp: (85443, 23)
Dimensions of y_test_imp: (85443,)
```

In [63]:
```
# Fit classifier to training set
clf = clf.fit(X_train_imp, y_train, sample_weight=sample_weight)
```

```
building tree 169 of 200
building tree 170 of 200
building tree 171 of 200
building tree 172 of 200
building tree 173 of 200
building tree 174 of 200
building tree 175 of 200
building tree 176 of 200
building tree 177 of 200
building tree 178 of 200
building tree 179 of 200
building tree 180 of 200
building tree 181 of 200
building tree 182 of 200
building tree 183 of 200
building tree 184 of 200
building tree 185 of 200
building tree 186 of 200
building tree 187 of 200
building tree 188 of 200
```

In [64]:
```
# Compute training metrics
accuracy = clf.score(X_train_imp, y_train)

#  Predict labels of test set
train_pred = clf.predict(X_train_imp)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_train, train_pred)
conf_mat = confusion_matrix(y_train.round(), train_pred.round())
clas_rep = classification_report(y_train.round(), train_pred.round())

# Print reports
print('{:=^80}'.format('New RF training report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.2s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    1.0s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    1.4s finished
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.1s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    1.0s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    1.3s finished

=============================New RF training report============================
=
Accuracy: 1.0000
MSE: 0.0000
Confusion matrix:
[[199021      0]
 [     0    343]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199021
           1       1.00      1.00      1.00       343

    accuracy                           1.00    199364
   macro avg       1.00      1.00      1.00    199364
weighted avg       1.00      1.00      1.00    199364
```

In [65]:
```python
# Compute testing metrics
accuracy = clf.score(X_test_imp, y_test)

# Predict labels of test set
y_pred = clf.predict(X_test_imp)

# Compute MSE, confusion matrix, classification report
mse = mean_squared_error(y_test, y_pred)
conf_mat = confusion_matrix(y_test.round(), y_pred.round())
clas_rep = classification_report(y_test.round(), y_pred.round())

# Print reports
print('{:=^80}'.format('New RF testing report'))
print('Accuracy: %.4f' % accuracy)
print("MSE: %.4f" % mse)
print("Confusion matrix:\n{}".format(conf_mat))
print("Classification report:\n{}".format(clas_rep))
```

```
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.0s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    0.4s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    0.5s finished
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.0s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    0.3s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    0.4s finished

=============================New RF testing report=============================
=
Accuracy: 0.9995
MSE: 0.0005
Confusion matrix:
[[85287     7]
 [   37   112]]
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85294
           1       0.94      0.75      0.84       149

    accuracy                           1.00     85443
   macro avg       0.97      0.88      0.92     85443
weighted avg       1.00      1.00      1.00     85443
```

In [66]:
```python
# Compute predicted probabilities
y_pred_prob = clf.predict_proba(X_test_imp)[:,1]
```

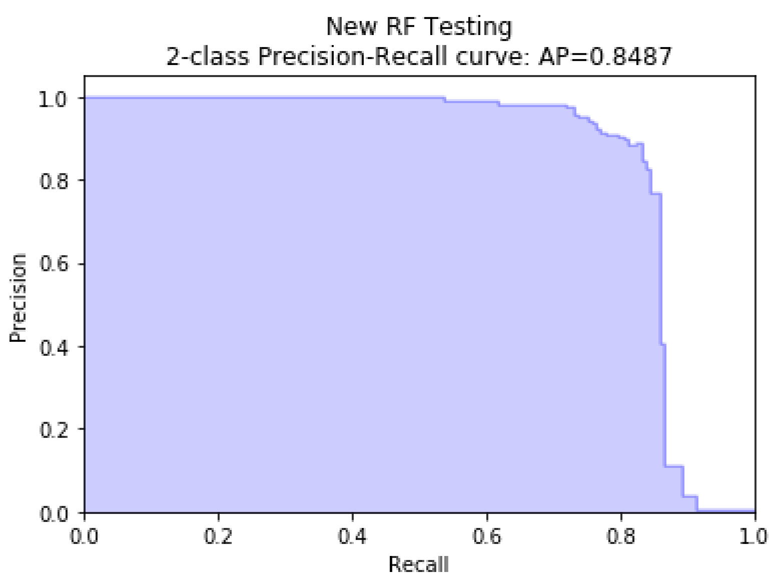
```
[Parallel(n_jobs=7)]: Using backend ThreadingBackend with 7 concurrent workers.
[Parallel(n_jobs=7)]: Done  27 tasks      | elapsed:    0.0s
[Parallel(n_jobs=7)]: Done 148 tasks      | elapsed:    0.3s
[Parallel(n_jobs=7)]: Done 200 out of 200 | elapsed:    0.5s finished
```

In [67]:
```
# Compute AUPRC score
average_precision = average_precision_score(y_test, y_pred_prob)
print("AUPRC: {}".format(average_precision))

# Plot PR curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
step_kwargs = ({'step': 'post'}
               if 'step' in signature(plt.fill_between).parameters
               else {})
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('New RF Testing\n2-class Precision-Recall curve: AP={0:0.4f}'.format(a
plt.show()
```

```
AUPRC: 0.8486548057918583
```



The testing AUPRC has not decreased after retraining the RF with the 24 most important of 29 features. We now have a more cost-effective RF classifier that has the same predictive power, but requires less data to train. These are the final scores:

Classifier	AUPRC
Random Forest (24 most important features)	0.8487
Random Forest (all 29 features)	0.8456
Kernel Support Vector Classifier	0.8081
Logistic Regression	0.7822
Stochastic Boosted Regression	0.7164

Conclusion

- Conventional metrics for classification models like accuracy and mean squared error give a too optimistic impression of model performance. Because the transaction dataset is a highly unbalanced dataset (anomalous transactions accounted for 0.1727% of all transactions), the area under Precision-Recall curve (AUPRC) is a more useful metric of model performance.
- The Random Forest provided the most cost-effective anomaly detection in terms of precision and recall (AUPRC=0.8487) and computation time, followed by the fast Logistic Regression (AUPRC=0.7822) and slower Kernel Support Vector Classifier (AUPRC=0.8081) and Stochastic Gradient Boosting Machine (AUPRC=0.7164).
- 5 of the original 29 features can be dropped without loss of model performance.