CASE STUDY - PHASE 3

Name	Andrew ID
Bharat Madan	bmadan
Dhwani Panjwani	dpanjwan
Sahil Ahuja	sahilahu
Sujai Chaudhary	sujaic

1.

i) For our model training we first use the prepare_data function to prepare the data. This process involves downsizing the data using the parameters date_range and keep only those columns which were selected in Phase2. Then it splits the data into X_test, y_test, X_train and y_train. We fit our classification models using the function fit_classification which uses accuracy, precision, recall, f-1 score and AUC score to evaluate the models. We will be using weighted average F-1 score as our metric of choice as it gives equal importance to both precision and recall also handles class imbalance.

ii)

Model	**Hyperparameters**
Random Classifier	strategy ('most_frequent', 'prior', 'stratified', 'uniform', 'constant')
Naive Bayes	var_smoothing (1e-10, 1e-5, 1e-3, 1e-1, 1)
L1 regularized logistic regression	C (0.01, 0.1, 1.0, 10.0, 100.0) and max_iter (100, 500, 1000)
L2 regularized logistic regression	C (0.01, 0.1, 1.0, 10.0, 100.0) and max_iter (100, 500, 1000)
Decision Tree	max_depth (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
Random Forest	n_estimators (100, 200, 300, 100) and max_depth (80, 90, 100, 110)
Multi Layer Perceptron	hidden_layer_sizes(20,40,60,80,100,120,140), activation ('identity', 'logistic', 'tanh', 'relu'),solver ('lbfgs', 'sgd', 'adam'] and learning_rate ('constant','adaptive')
K Neighbors Classifier	n_neighbors (1,2,4,8,16,32,64,128)
Adaboost	n_estimators (10,20,30,40,50,60,70,100)

iii) Performance measures used - Accuracy, F1 score, precision, recall and AUC

	Accuracy	F1	Precision	Recall	AUC
Random Classifier	0.80395	0.7166	0.6463	0.8040	0.50
Naive Bayes	0.804	0.7176	0.7493	0.8040	0.67
L1 regularized logistic regression	0.805	0.7252	0.7598	0.8050	0.69
L2 regularized logistic regression	0.80415	0.7199	0.7514	0.8042	0.69
Decision Tree	0.80395	0.7166	0.6463	0.8040	0.58
Random Forest	0.80385	0.7199	0.7426	0.8038	0.68
Multi Layer Perceptron	0.8049	0.7285	0.7555	0.8049	0.70
K Neighbors Classifier	0.8037	0.7190	0.7367	0.8037	0.67
Adaboost	0.80345	0.7197	0.7332	0.8034	0.69

2.

Both random sampling and temporal train/test split has some advantages and disadvantages based on the data.

Random Sampling - It would work well if the patterns are distributed evenly over the dataset and would help in improving generalization by removing any bias. But in this case random sampling might not work well as the data keeps changing over time and the patter are not uniform across the dataset.

Temporal Splitting - It works well with financial data as financial data keeps changing over time. It can also help in detecting biases or changes in data distribution over time. This would be a better model for our use case because the status of a loan keeps changin over time. We can model how the status of loan changes over time and use it to predict whether the loan defaults or not. One disadvantage would be that it may exclude patters which occurs after cut-off time. This can lead to overfitting.

3.

i)

Lending Club derived features: grade, int_rate, verification_status
Features which are correlated with or affect the LC derived features:
fico_range_low, fico_range_high, annual_inc, home_ownership,dti,
Emp_length

ii) We are choosing weighted average F-1 score as our metric of our choice.
L1 logistic regression achieved a F-1 score of 0.7165 when trained with 'grade' as the only input feature.

iii)

Average performance +- std of Gaussian NB classifier without LC derived feature: 0.7179318482238959
0.7179318482238957
Average performance +- std of L1 Logistic Regression without LC derived feature: 0.7200933105064655
0.7200551986529892
Average performance +- std of L2 Logistic Regression without LC derived feature: 0.7187308907190054
0.7187308907190048
Average performance +- std of Decision Tree classifier without LC derived feature: 0.7165781784417529
0.7165781784417529

Average performance +- std of Random classifier without LC derived feature: 0.7199138126769689 0.7179140588273736
Average performance +- std of MLP classifier without LC derived feature: 0.7210555402547913 0.7178019790772292
Average performance +- std of AdaBoost classifier without LC derived feature: 0.7166980148598092 0.716698014859809

Even after removing 'grade' we are getting similar performance on all the models. This suggests that there are other features which are heavily correlated with 'grade' or affect it in some way.

4.
The best model according to part 3 is Logistic Regression with L1 regularization. For that according to kendall tau metric we are getting a similarity score 46% which shows that the grades assigned by Lending Club match our model's scores 46% of the time.

5.
The model trained on 2010 data achieved a weighted average F-1 score of 69.3% on 2018 data. The model trained on 2017 data achieved a weighted average F-1 score of 68.26% on 2018 data. Since both the scores are similar we can conclude that there is no major difference in the underlying distribution of data over the years and our model is time stable.

6. We notice that there is a slight increase in performance when using the old features. Even though the old feature set had many irrelevant features like zip-code,id, etc. the model was able to achieve slightly better performance than the models we had fitted earlier. The surprising observation here is that these supposedly irrelevant features didn't add any noise to the model.

7. The following are the R2

Regressing against all returns

Model	M1	M2	M3(2.3%)	M3(4.0%)
L1 regressor	0.07	0.009	0.0018	0.017
L2 regressor	0.1	0.016	0.036	0.033
Neural Network regressor	0.09	0.016	0.03	0.02
Random Forest Regressor	0.1	0.019	0.04	0.037

From eyeballing the above, we can say that a random forest regressor gives the best results.

Regressing against returns for defaulted loans

Model	M1	M2	M3(2.3%)	M3(4.0%)
L1 regressor	0.1	0.01	-0.0002	0.03
L2 regressor	0.125	0.019	0.044	0.058
Neural Network regressor	0.122	0.019	0.04	0.039
Random Forest Regressor	0.13	0.023	0.062	0.074

From eyeballing the above, we can say that a random forest regressor gives the best results.

Regressing against returns for non defaulted loans

Model	M1	M2	M3(2.3%)	M3(4.0%)
L1 regressor	0.0375	0.01	-2.51e-05	-5.3e-06
L2 regressor	0.08	0.016	0.032	0.03
Neural Network regressor	0.08	0.017	0.026	0.0215
Random Forest Regressor	0.0944	0.028	0.028	0.029

From eyeballing the above, we can say that a random forest regressor gives the best results.

8. (i) and (ii)

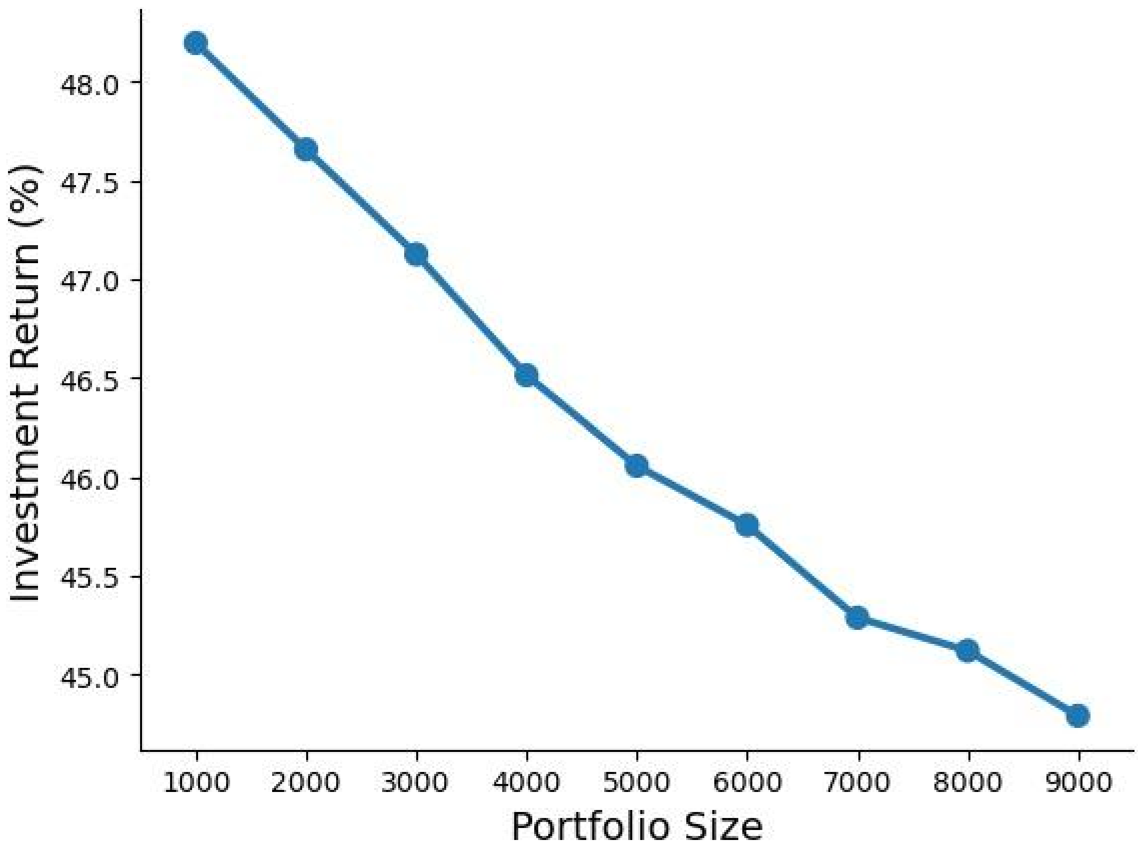
Strategy	M1	M2	M3(2.3%)	M3(4.0%)
Rand	0.399	1.48	0.499	1.43
Def	0.402	1.65	0.49	1.42
Ret	0.5	1.58	0.49	1.42
DefRet	0.36	1.26	0.42	1.269
BEST	Ret	Def	Rand	Rand

According to the table, Return strategy performs best. The random strategy performs best in M3, however it is not advisable to use it since it randomly picks loans to invest in instead of regressing. In M2, the main issue is that if a loan defaults early, annualizing the loss can result in a huge over-estimate of the negative return. However, in M2, we choose Default as the best strategy and not random, thus it doesn't cost loss.
In the case of M1, Ret and Def perform slightly better than random.
In case of M2 too, Ret and Def perform slightly better than random.
Lastly, in case of M3 (both cases), random strategy appears to perform better than the data driven strategies.

9. We observe a negative correlation between portfolio size and investment return, i.e. as the portfolio size increases, the return on the investment decreases. This is because, as the portfolio size increases, the total amount invested in the loan (f) increases. This decreases the return on investment according to the formula:

(p-f)/f * 12/t

Since this method supposes that, once the loan is paid back, the
the investor is forced to sit with the money without reinvesting it anywhere else, the larger the portfolio size, the smaller the portfolio size, the higher the return, as it means that it is invested only in a single space.



Phase 3 - Modeling

Note 1: the following starting code only generates a single random train/test split when default_seed is used. You need to modify the code to generate 100 independent train/test splits with different seeds and report the average results on those independent splits along with standard deviation.

Note 2: You are completely free to use your own implementation.

In [111…

```
# Load general utilities
# ----------------------
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.axes as ax
import datetime
import numpy as np
import pickle
import time
import seaborn as sns

# Load sklearn utilities
# ----------------------
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.model_selection import GridSearchCV

from sklearn.metrics import accuracy_score, classification_report, roc_auc_score, roc_curve, brier_score_loss, mean_squ
from sklearn.metrics import f1_score
from sklearn.calibration import calibration_curve

# Load classifiers
# ----------------
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import RidgeClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
```

```
from sklearn.dummy import DummyClassifier

# Other Packages
# --------------
from scipy.stats import kendalltau
from sklearn.neural_network import MLPRegressor
from sklearn import linear_model
from sklearn.ensemble import RandomForestRegressor
from sklearn.cluster import KMeans
from six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
# from scipy.interpolate import spline
from scipy.interpolate import interp1d
# Load debugger, if required
#import pixiedust
pd.options.mode.chained_assignment = None #'warn'

# suppress all warnings
import warnings
warnings.filterwarnings("ignore")
```

In [2]:

```
# Define a function that, given a CVGridSearch object, finds the
# percentage difference between the best and worst scores
def find_score_variation(cv_model):
    all_scores = cv_model.cv_results_['mean_test_score']
    return( np.abs((max(all_scores) - min(all_scores))) * 100 / max(all_scores) )

    '''
    which_min_score = np.argmin(all_scores)

    all_perc_diff = []

    try:
        all_perc_diff.append( np.abs(all_scores[which_min_score - 1] - all_scores[which_min_score])*100 / min(all_score
    except:
        pass

    try:
        all_perc_diff.append( np.abs(all_scores[which_min_score + 1] - all_scores[which_min_score])*100 / min(all_score
    except:
        pass

    return ( np.mean(all_perc_diff) )
    '''
```

```
# Define a function that checks, given a CVGridSearch object,
# whether the optimal parameters lie on the edge of the search
# grid
def find_opt_params_on_edge(cv_model):
    out = False

    for i in cv_model.param_grid:
        if cv_model.best_params_[i] in [ cv_model.param_grid[i][0], cv_model.param_grid[i][-1] ]:
            out = True
            break

    return out
```

Define a default random seed and an output file

In [3]:
```
default_seed = 1
output_file = "output_sample"
```

In [4]:
```
# Create a function to print a line to our output file

def dump_to_output(key, value):
    with open(output_file, "a") as f:
        f.write(",".join([str(default_seed), key, str(value)]) + "\n")
```

Load the data and engineer the features

In [7]:
```
# Read the data and features from the pickle file saved in CS-Phase 2
data, discrete_features, continuous_features, ret_cols = pickle.load( open( "2003_download/clean_data.pickle", "rb" ) )
```

In [8]:
```
## Create the outcome columns: True if loan_status is either Charged Off or Default, False otherwise
data["outcome"] = data["loan_status"] == ('Charged Off' or "Default")
```

In [9]:
```
# Create a feature for the length of a person's credit history at the time the loan is issued
data['cr_hist'] = (data.issue_d - data.earliest_cr_line) / np.timedelta64(1, 'M')
continuous_features.append('cr_hist')
```

In [10]:
```
# Randomly assign each row to a training and test set. We do this now because we will be fitting a variety of models on
np.random.seed(default_seed)
```

```
## create the train columns where the value is True if it is a train instance and False otherwise. Hint: use np.random.
data['train'] = np.random.choice([True,False],size = len(data),p=[.7,.3])
```

In [11]:

```
# Create a matrix of features and outcomes, with dummies. Record the names of the dummies for later use
X_continuous = data[continuous_features].values

X_discrete = pd.get_dummies(data[discrete_features], dummy_na = True, prefix_sep = "::", drop_first = True)
discrete_features_dummies = X_discrete.columns.tolist()
X_discrete = X_discrete.values

X = np.concatenate( (X_continuous, X_discrete), axis = 1 )

y = data.outcome.values

train = data.train.values
```

Prepare functions to fit and evaluate models

In [70]:

```
def prepare_data(data_subset = np.array([True]*len(data)),
                    n_samples_train = 30000,
                    n_samples_test = 20000,
                    feature_subset = None,
                    date_range_train = (data.issue_d.min(), data.issue_d.max()),
                    date_range_test = (data.issue_d.min(), data.issue_d.max()),
                    random_state = default_seed):
    '''
    This function will prepare the data for classification or regression.
    It expects the following parameters:
      - data_subset: a numpy array with as many entries as rows in the
                     dataset. Each entry should be True if that row
                     should be used, or False if it should be ignored
      - n_samples_train: the total number of samples to be used for training.
                         Will trigger an error if this number is larger than
                         the number of rows available after all filters have
                         been applied
      - n_samples_test: as above for testing
      - feature_subect: A list containing the names of the features to be
                        used in the model. In None, all features in X are
                        used
      - date_range_train: a tuple containing two dates. All rows with loans
                          issued outside of these two dates will be ignored in
                          training
      - date_range_test: as above for testing
```

```
      - random_state: the random seed to use when selecting a subset of rows

    Note that this function assumes the data has a "Train" column, and will
    select all training rows from the rows with "True" in that column, and all
    the testing rows from those with a "False" in that column.

    This function returns a dictionary with the following entries
      - X_train: the matrix of training data
      - y_train: the array of training labels
      - train_set: a Boolean vector with as many entries as rows in the data
                  that denotes the rows that were used in the train set
      - X_test: the matrix of testing data
      - y_test: the array of testing labels
      - test_set: a Boolean vector with as many entries as rows in the data
                  that denotes the rows that were used in the test set
    '''

    np.random.seed(random_state)

    # Filter down the data to the required date range, and downsample
    # as required
    filter_train = ( train & (data.issue_d >= date_range_train[0]) &
                            (data.issue_d <= date_range_train[1]) & data_subset ).values
    filter_test = ( (train == False) & (data.issue_d >= date_range_test[0])
                            & (data.issue_d <= date_range_test[1]) & data_subset ).values

    #print(filter_train.sum())

    filter_train[ np.random.choice( np.where(filter_train)[0], size = filter_train.sum()
                                               - n_samples_train, replace = False ) ] = False
    filter_test[ np.random.choice( np.where(filter_test)[0], size = filter_test.sum()
                                               - n_samples_test, replace = False ) ] = False

    # Prepare the training and test set
    X_train = X[ filter_train , :]
    X_test = X[ filter_test, :]
    if feature_subset != None:
        cols = [i for i, j in enumerate(continuous_features + discrete_features_dummies)
                                                if j.split("::")[0] in feature_subset]
        X_train = X_train[ : , cols ]
        X_test = X_test[ : , cols ]

    y_train = y[ filter_train ]
    y_test = y[ filter_test ]
```

```
    # Scale the variables
    scaler = preprocessing.MinMaxScaler()

    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # return training and testing data
    out = {'X_train':X_train, 'y_train':y_train, 'train_set':filter_train,
           'X_test':X_test, 'y_test':y_test, 'test_set':filter_test}

    return out
```

In [100…

```
def fit_classification(model, data_dict,
                          cv_parameters = {},
                           model_name = None,
                          random_state = default_seed,
                          output_to_file = True,
                          print_to_screen = True):
    '''
    This function will fit a classification model to data and print various evaluation
    measures. It expects the following parameters
      - model: an sklearn model object
      - data_dict: the dictionary containing both training and testing data;
                   returned by the prepare_data function
      - cv_parameters: a dictionary of parameters that should be optimized
                       over using cross-validation. Specifically, each named
                       entry in the dictionary should correspond to a parameter,
                       and each element should be a list containing the values
                       to optimize over
      - model_name: the name of the model being fit, for printouts
      - random_state: the random seed to use
      - output_to_file: if the results will be saved to the output file
      - print_to_screen: if the results will be printed on screen

    If the model provided does not have a predict_proba function, we will
    simply print accuracy diagnostics and return.

    If the model provided does have a predict_proba function, we first
    figure out the optimal threshold that maximizes the accuracy and
    print out accuracy diagnostics. We then print an ROC curve, sensitivity/
    specificity curve, and calibration curve.

    This function returns a dictionary with the following entries
      - model: the best fitted model
      - y_pred: predictions for the test set
```

```
      - y_pred_probs: probability predictions for the test set, if the model
                      supports them
      - y_pred_score: prediction scores for the test set, if the model does not
                      output probabilities.
    '''

    np.random.seed(random_state)

    # -------------------------
    #   Step 1 - Load the data
    # -------------------------
    X_train = data_dict['X_train']
    y_train = data_dict['y_train']

    X_test = data_dict['X_test']
    y_test = data_dict['y_test']

    filter_train = data_dict['train_set']

    # -------------------------
    #   Step 2 - Fit the model
    # -------------------------

    cv_model = GridSearchCV(model, cv_parameters)

    start_time = time.time()
    cv_model.fit(X_train, y_train)
    end_time = time.time()

    best_model = cv_model.best_estimator_

    if print_to_screen:

        if model_name != None:
            print("==========================================================")
            print("  Model: " + model_name)
            print("==========================================================")

        print("Fit time: " + str(round(end_time - start_time, 2)) + " seconds")
        print("Optimal parameters:")
        print(cv_model.best_params_)
        print("")

    # ------------------------------
    #   Step 3 - Evaluate the model
```

```
    # ------------------------------

    # If possible, make probability predictions
    try:
        y_pred_probs = best_model.predict_proba(X_test)[:,1]
        fpr, tpr, thresholds = roc_curve(y_test, y_pred_probs)

        probs_predicted = True
    except:
        probs_predicted = False

    # Make predictions; if we were able to find probabilities, use
    # the threshold that maximizes the accuracy in the training set.
    # If not, just use the learner's predict function
    if probs_predicted:
        y_train_pred_probs = best_model.predict_proba(X_train)[:,1]
        fpr_train, tpr_train, thresholds_train = roc_curve(y_train, y_train_pred_probs)

        true_pos_train = tpr_train*(y_train.sum())
        true_neg_train = (1 - fpr_train) *(1-y_train).sum()

        best_threshold_index = np.argmax(true_pos_train + true_neg_train)
        best_threshold = 1 if best_threshold_index == 0 else thresholds_train[ best_threshold_index ]

        if print_to_screen:
            print("Accuracy-maximizing threshold was: " + str(best_threshold))

        y_pred = (y_pred_probs > best_threshold)
    else:
        y_pred = best_model.predict(X_test)

    if print_to_screen:
        print("Accuracy: ", accuracy_score(y_test, y_pred))
        print(classification_report(y_test, y_pred, target_names =['No default', 'Default'], digits = 4))

    if print_to_screen:
        if probs_predicted:
            plt.figure(figsize = (13, 4.5))
            plt.subplot(2, 2, 1)

            plt.title("ROC Curve (AUC = %0.2f)"% roc_auc_score(y_test, y_pred_probs))
            plt.plot(fpr, tpr, 'b')
            plt.plot([0,1],[0,1],'r--')
            plt.xlim([0,1]); plt.ylim([0,1])
            plt.ylabel('True Positive Rate')
```

```
        plt.xlabel('False Positive Rate')

        plt.subplot(2, 2, 3)

        plt.plot(thresholds, tpr, 'b', label = 'Sensitivity')
        plt.plot(thresholds, 1 -fpr, 'r', label = 'Specificity')
        plt.legend(loc = 'lower right')
        plt.xlim([0,1]); plt.ylim([0,1])
        plt.xlabel('Threshold')

        plt.subplot(2, 2, 2)

        fp_0, mpv_0 = calibration_curve(y_test, y_pred_probs, n_bins = 10)
        plt.plot([0,1], [0,1], 'k:', label='Perfectly calibrated')
        plt.plot(mpv_0, fp_0, 's-')
        plt.ylabel('Fraction of Positives')
        plt.xlim([0,1]); plt.ylim([0,1])
        plt.legend(loc ='upper left')

        plt.subplot(2, 2, 4)
        plt.hist(y_pred_probs, range=(0, 1), bins=10, histtype="step", lw=2)
        plt.xlim([0,1]); plt.ylim([0,20000])
        plt.xlabel('Mean Predicted Probability')
        plt.ylabel('Count')

        #plt.tight_layout()
        plt.show()

# Additional Score Check
if probs_predicted:
    y_train_score = y_train_pred_probs
else:
    y_train_score = best_model.decision_function(X_train)

tau, p_value = kendalltau(y_train_score, data.grade[filter_train])
if print_to_screen:
    print("")
    print("Similarity to LC grade ranking: ", tau)

if probs_predicted:
    brier_score = brier_score_loss(y_test, y_pred_probs)
    if print_to_screen:
        print("Brier score:", brier_score)

# Return the model predictions, and the
```

```
    # test set
    # -------------------------------------
    out = {'model':best_model, 'y_pred_labels':y_pred,"performance":f1_score(y_test, y_pred,average='weighted')}

    if probs_predicted:
        out.update({'y_pred_probs':y_pred_probs})
    else:
        y_pred_score = best_model.decision_function(X_test)
        out.update({'y_pred_score':y_pred_score})

    # Output results to file
    # ----------------------
    if probs_predicted and output_to_file:
        # Check whether any of the CV parameters are on the edge of
        # the search space
        opt_params_on_edge = find_opt_params_on_edge(cv_model)
        dump_to_output(model_name + "::search_on_edge", opt_params_on_edge)
        if print_to_screen:
            print("Were parameters on edge? : " + str(opt_params_on_edge))

        # Find out how different the scores are for the different values
        # tested for by cross-validation. If they're not too different, then
        # even if the parameters are off the edge of the search grid, we should
        # be ok
        score_variation = find_score_variation(cv_model)
        dump_to_output(model_name + "::score_variation", score_variation)
        if print_to_screen:
            print("Score variations around CV search grid : " + str(score_variation))

        # Print out all the scores
        dump_to_output(model_name + "::all_cv_scores", str(cv_model.cv_results_['mean_test_score']))
        if print_to_screen:
            print( str(cv_model.cv_results_['mean_test_score']) )

        # Dump the AUC to file
        dump_to_output(model_name + "::roc_auc", roc_auc_score(y_test, y_pred_probs) )

    return out
```

Train and Test different machine learning classification models

The machine learning models listed in the following are just our suggestions. You are free to try any other models that you would like to experiment with.

In [112…

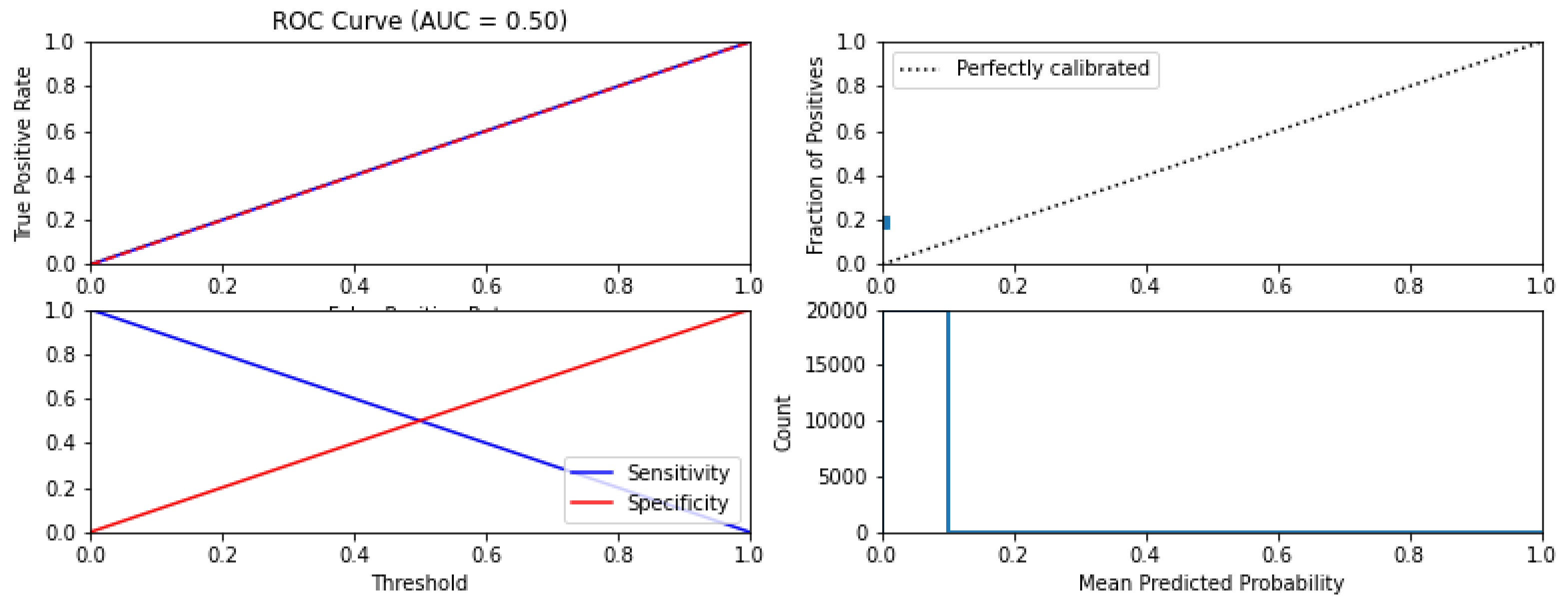
```
## define your set of features to use in different models
your_features = ['loan_amnt', 'funded_amnt', 'term', 'int_rate', 'grade',
       'emp_length', 'home_ownership', 'annual_inc', 'verification_status',
       'issue_d', 'loan_status', 'purpose', 'dti', 'delinq_2yrs',
       'earliest_cr_line', 'open_acc', 'pub_rec', 'fico_range_high',
       'fico_range_low', 'revol_bal', 'revol_util', 'total_pymnt',
       'recoveries', 'last_pymnt_d', 'loan_length', 'term_num', 'ret_PESS',
       'ret_OPT', 'ret_INTa', 'ret_INTb', 'cr_hist', 'train']
# prepare the train, test data for training models
data_dict = prepare_data(feature_subset = your_features)

all_features = pd.Series(continuous_features + discrete_features_dummies)
idx = [i for i, j in enumerate(continuous_features + discrete_features_dummies)
                                                    if j.split("::")[0] in your_features]
selected_features = all_features[idx]
selected_features.reset_index(drop=True,inplace=True)
```

DummyClassifier (Baseline)

In [113…

```
dummy_clf = DummyClassifier()
dummy=fit_classification(dummy_clf,data_dict,
                         cv_parameters = {'strategy': ["most_frequent", "prior", "stratified", "uniform", "constant"]},
                         model_name = 'Dummy Classifier')
```

```
=========================================================
  Model: Dummy Classifier
=========================================================
Fit time: 0.14 seconds
Optimal parameters:
{'strategy': 'most_frequent'}

Accuracy-maximizing threshold was: 1
Accuracy:  0.80395
              precision    recall  f1-score   support

  No default     0.8040    1.0000    0.8913     16079
     Default     0.0000    0.0000    0.0000      3921

    accuracy                         0.8040     20000
   macro avg     0.4020    0.5000    0.4457     20000
weighted avg     0.6463    0.8040    0.7166     20000
```




```
Similarity to LC grade ranking:  nan
Brier score: 0.19605
Were parameters on edge? : True
Score variations around CV search grid : 37.509866644509984
[0.80236667 0.80236667 0.68726667 0.5014            nan]
```

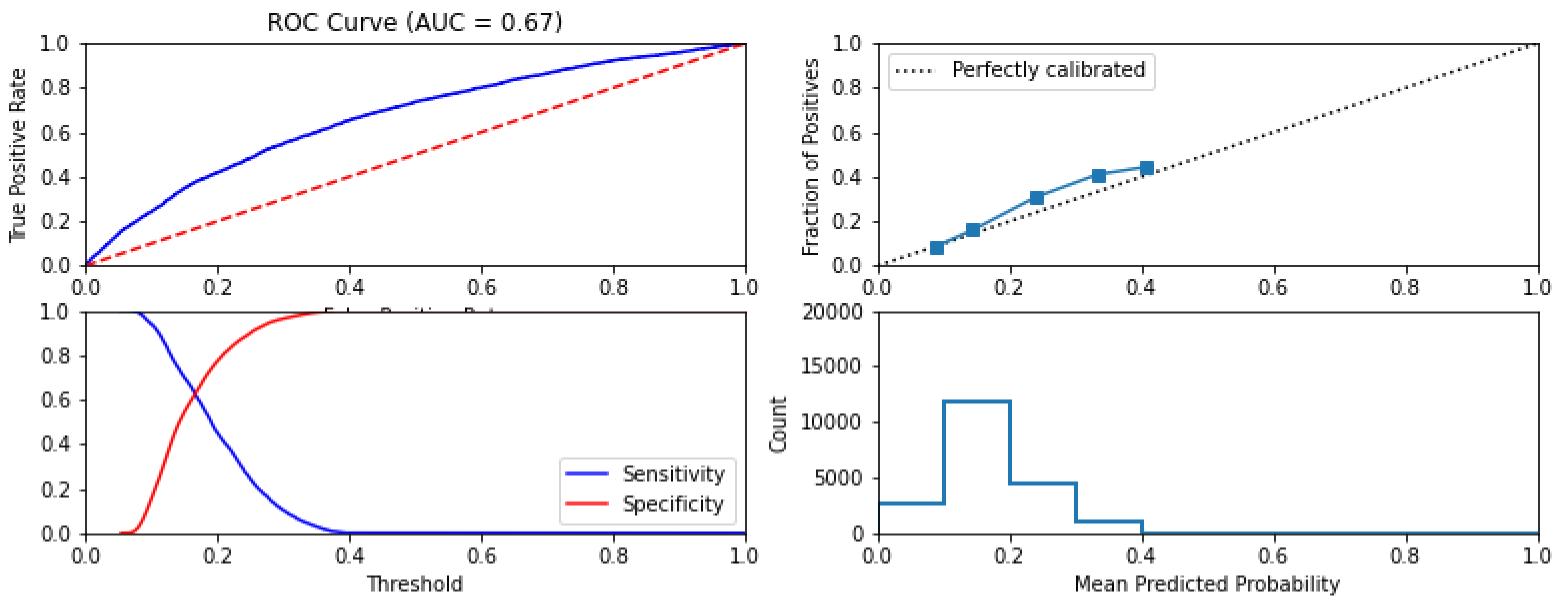
Naive Bayes

In [49]:

```
## Train and test a naive bayes classifier

gnb = GaussianNB()
gnb = fit_classification(gnb,data_dict,
                         cv_parameters = {'var_smoothing': [1e-10, 1e-5, 1e-3, 1e-1, 1]},
                         model_name = 'Gaussian Naive Bayes')
```

```
============================================================
  Model: Gaussian Naive Bayes
============================================================
Fit time: 1.03 seconds
Optimal parameters:
{'var_smoothing': 1}

Accuracy-maximizing threshold was: 0.39119408294655195
Accuracy:  0.804
```

	precision	recall	f1-score	support
No default	0.8043	0.9994	0.8913	16079
Default	0.5238	0.0028	0.0056	3921
accuracy			0.8040	20000
macro avg	0.6641	0.5011	0.4484	20000
weighted avg	0.7493	0.8040	0.7176	20000

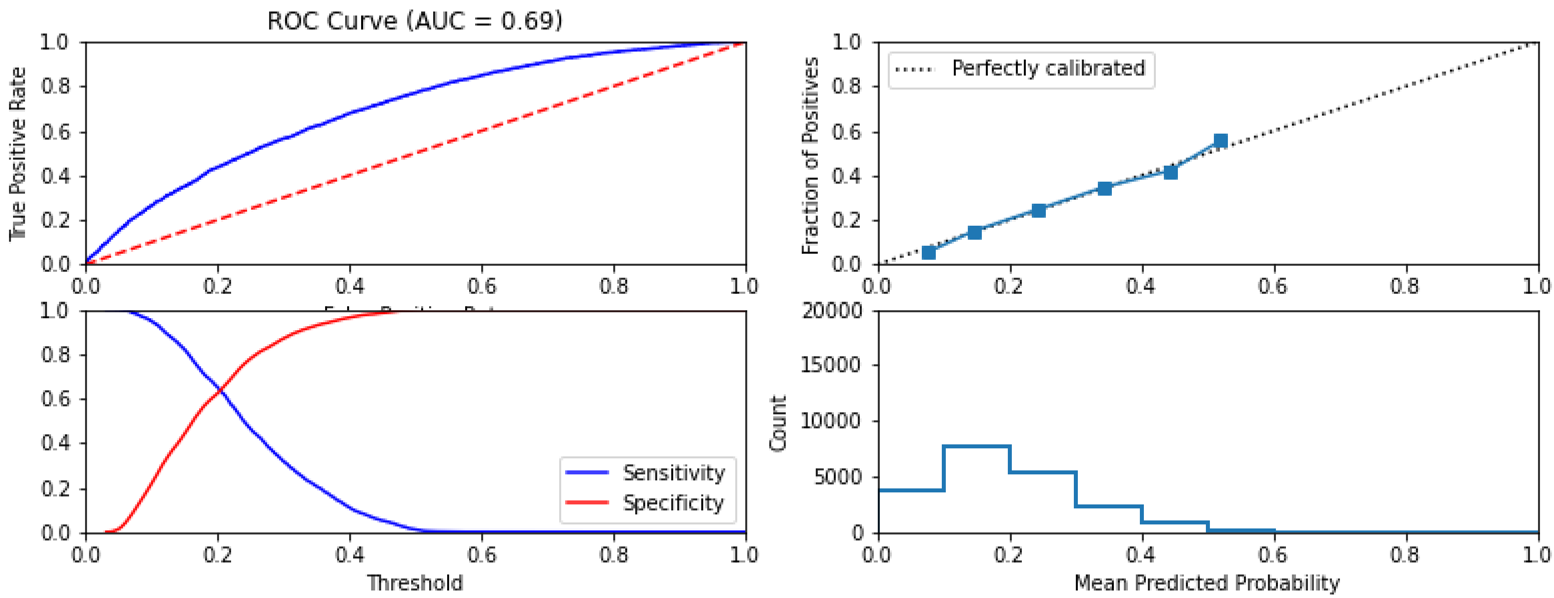



```
Similarity to LC grade ranking:  0.5913431556731094
Brier score: 0.15024007375297854
Were parameters on edge? : True
Score variations around CV search grid : 60.358938141331905
[0.31806667 0.5355     0.74406667 0.77026667 0.80236667]
```

l_1 regularized logistic regression

In [51]:
```
## Train and test a l_1 regularized logistic regression classifier

l1_logistic = LogisticRegression(penalty='l1',solver = 'liblinear')
cv_parameters = {'C': [0.01, 0.1, 1.0, 10.0, 100.0],
                 'max_iter': [100, 500, 1000]}

l1_logistic = fit_classification(l1_logistic,
                                 data_dict,
                                 cv_parameters =cv_parameters ,
                                 model_name ="l_1 regularized logistic regression classifier" )
```

```
=========================================================
  Model: l_1 regularized logistic regression classifier
=========================================================
Fit time: 127.29 seconds
Optimal parameters:
{'C': 0.1, 'max_iter': 100}

Accuracy-maximizing threshold was: 0.4851972981709148
Accuracy:  0.805
              precision    recall  f1-score   support

  No default     0.8069    0.9958    0.8914     16079
     Default     0.5669    0.0227    0.0436      3921

    accuracy                         0.8050     20000
   macro avg     0.6869    0.5092    0.4675     20000
weighted avg     0.7598    0.8050    0.7252     20000
```




```
Similarity to LC grade ranking:  0.7762085980352409
Brier score: 0.14612249171148836
Were parameters on edge? : True
Score variations around CV search grid : 0.09550701768956399
[0.80236667 0.80236667 0.80236667 0.80273333 0.8027     0.80273333
 0.8021     0.80213333 0.80213333 0.80203333 0.80203333 0.80196667
 0.802      0.80203333 0.802     ]
```

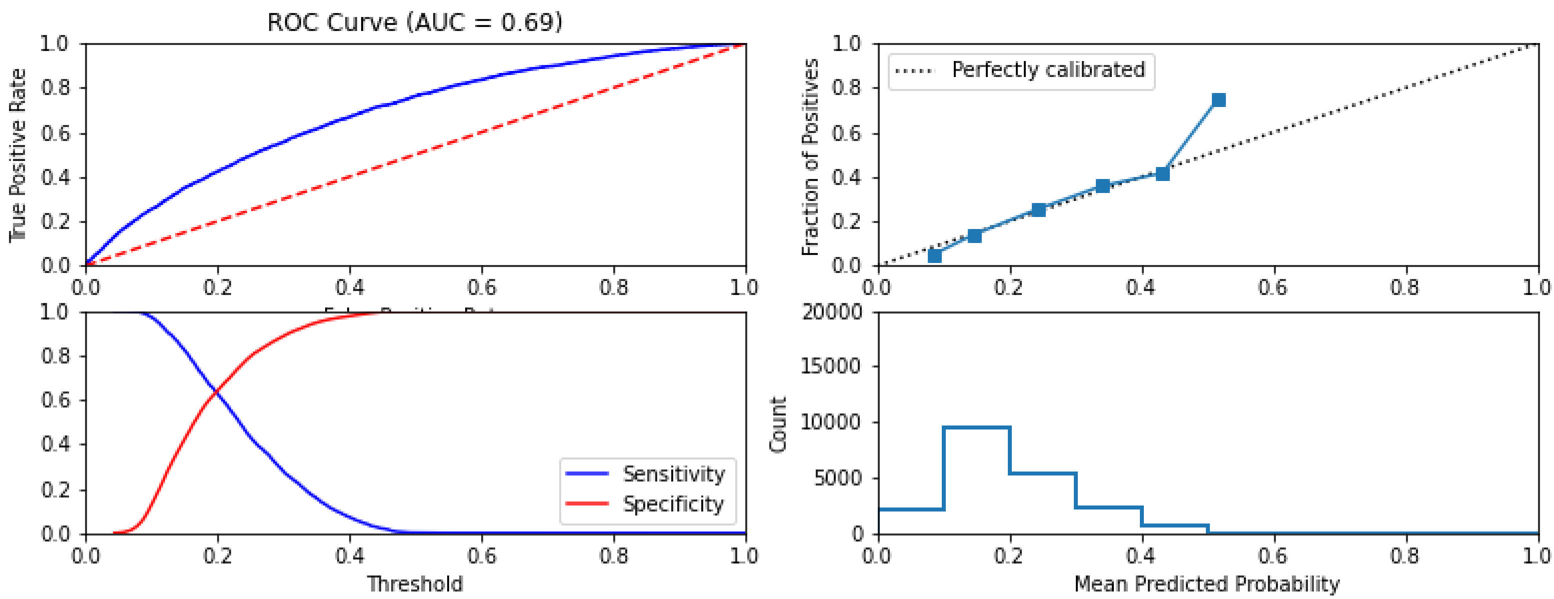
l_2 regularized logistic regression

In [52]:

```
## Train and test a l_2 regularized logistic regression classifier

l2_logistic = LogisticRegression(penalty='l2')
cv_parameters = {'C': [0.01, 0.1, 1.0, 10.0, 100.0],
                 'max_iter': [100, 500, 1000]}

l2_logistic = fit_classification(l2_logistic,
                                 data_dict,
                                 cv_parameters =cv_parameters ,
                                 model_name ="l_2 regularized logistic regression classifier" )
```

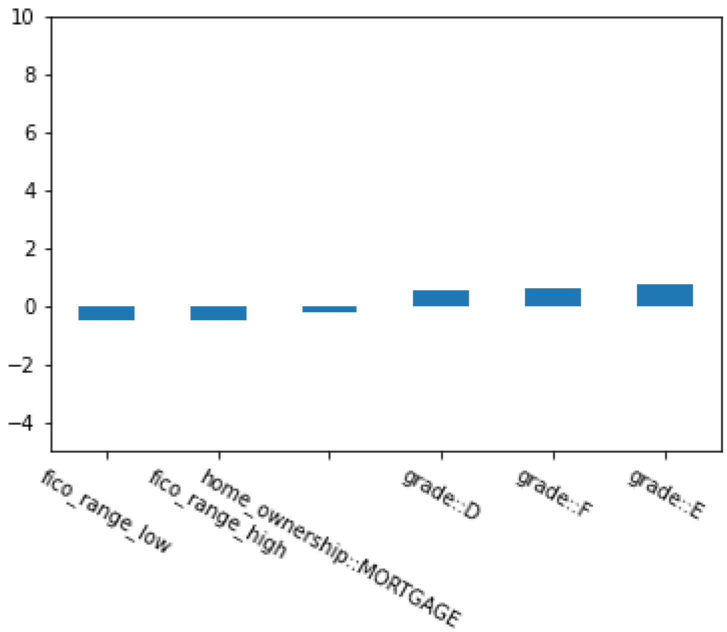
```
==========================================================
  Model: l_2 regularized logistic regression classifier
==========================================================
Fit time: 28.13 seconds
Optimal parameters:
{'C': 0.01, 'max_iter': 100}

Accuracy-maximizing threshold was: 0.4677875888163105
Accuracy:  0.80415
```

	precision	recall	f1-score	support
No default	0.8050	0.9981	0.8912	16079
Default	0.5312	0.0087	0.0171	3921
accuracy			0.8042	20000
macro avg	0.6681	0.5034	0.4542	20000
weighted avg	0.7514	0.8042	0.7199	20000




```
Similarity to LC grade ranking:  0.6751908257684053
Brier score: 0.147250921277263
Were parameters on edge? : True
Score variations around CV search grid : 0.08722741433020846
[0.8025     0.8025     0.8025     0.80246667 0.80246667 0.80246667
 0.80203333 0.80203333 0.80203333 0.8018     0.8023     0.8023
 0.8019     0.80226667 0.80226667]
```

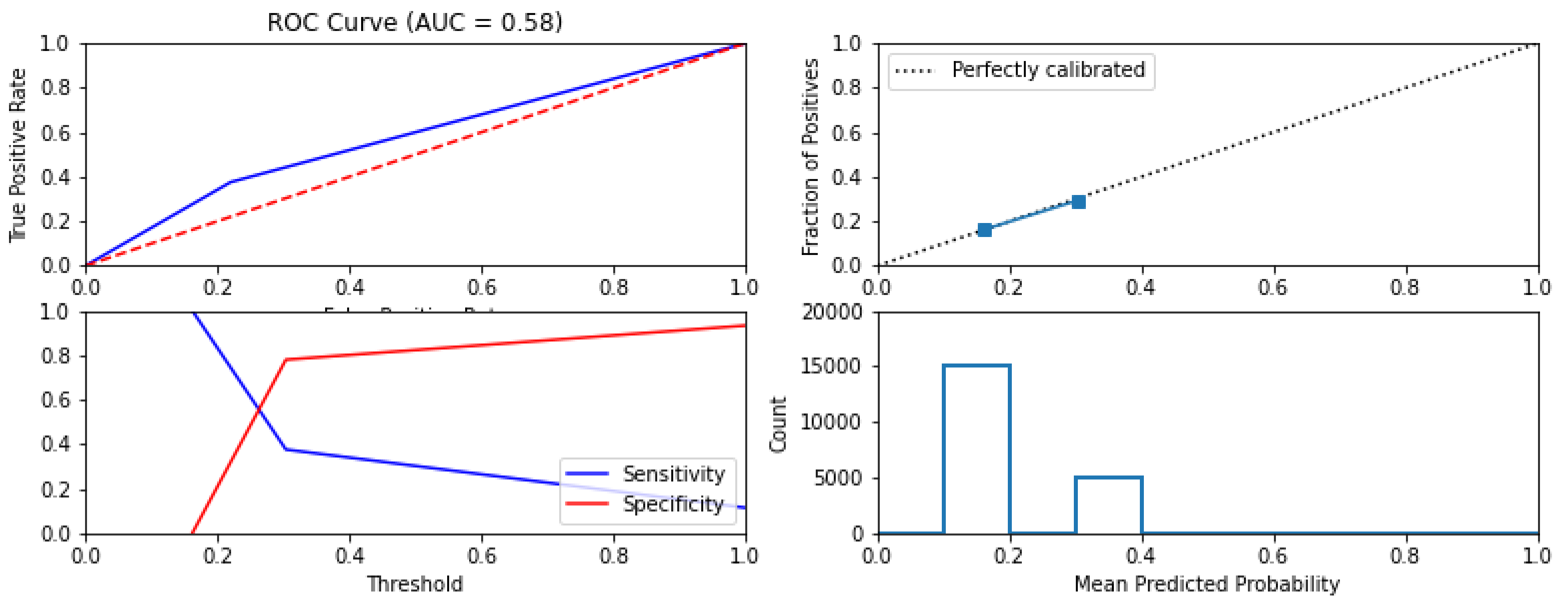
In [53]:
```
## plot top 3 features with the most positive (and negative) weights
top_and_bottom_idx = list(np.argsort(l2_logistic['model'].coef_)[0,:3]) + list(np.argsort(l2_logistic['model'].coef_)[0
bplot = pd.Series(l2_logistic['model'].coef_[0,top_and_bottom_idx])
xticks = selected_features[top_and_bottom_idx]
p1 = bplot.plot(kind='bar',rot=-30,ylim=(-5,10))
p1.set_xticklabels(xticks)
plt.show()
```



Decision tree

In [54]:
```
## Train and test a decision tree classifier

decision_tree = DecisionTreeClassifier()
cv_parameters = {'max_depth':np.linspace(1, 20, 20)}
data_dict = prepare_data(feature_subset = your_features)

decision_tree = fit_classification(decision_tree,data_dict,cv_parameters =cv_parameters ,model_name ="Decision Tree" )
```

```
===========================================================
  Model: Decision Tree
===========================================================
Fit time: 17.05 seconds
Optimal parameters:
{'max_depth': 1.0}

Accuracy-maximizing threshold was: 1
Accuracy:  0.80395
              precision    recall  f1-score   support

  No default     0.8040    1.0000    0.8913     16079
     Default     0.0000    0.0000    0.0000      3921

    accuracy                         0.8040     20000
   macro avg     0.4020    0.5000    0.4457     20000
weighted avg     0.6463    0.8040    0.7166     20000
```

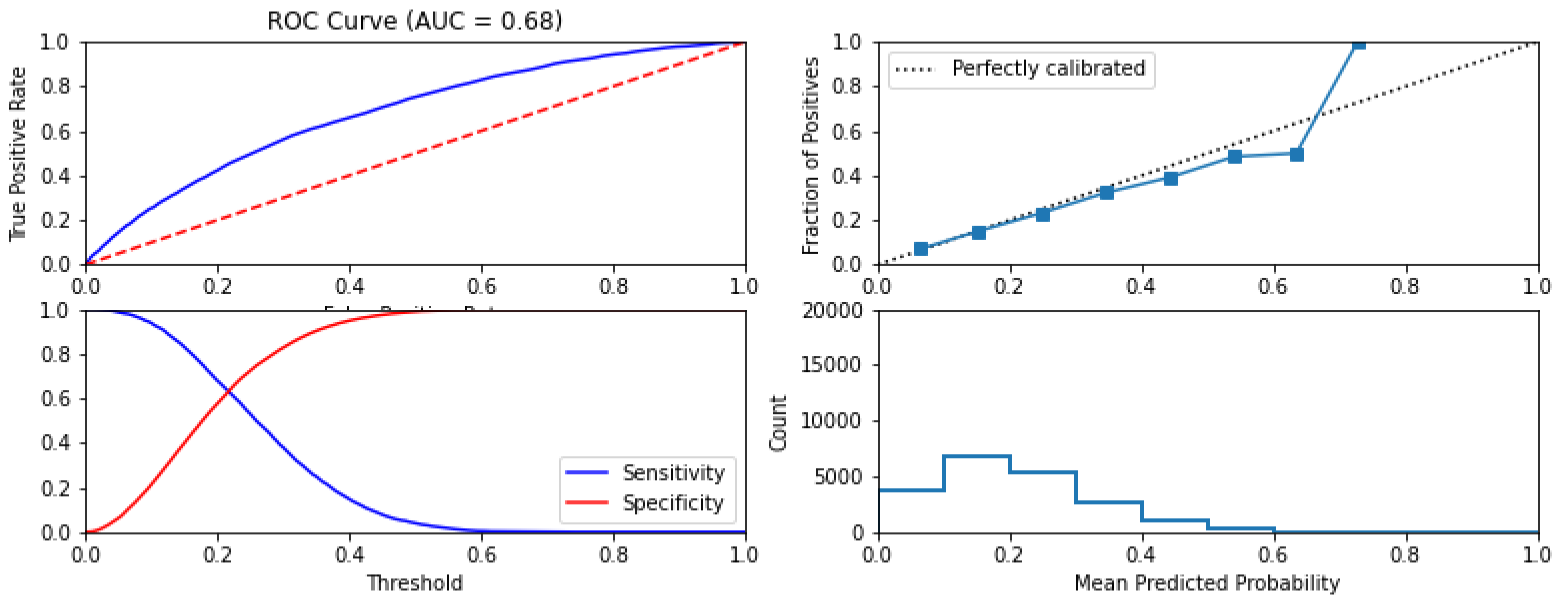



```
Similarity to LC grade ranking:  0.34812525840592184
Brier score: 0.15442823059127062
Were parameters on edge? : True
Score variations around CV search grid : 9.704623821195625
[0.80236667 0.80236667 0.80236667 0.8017     0.8001     0.79843333
 0.7971     0.79426667 0.78993333 0.7822     0.77843333 0.77126667
 0.76713333 0.7604     0.7529     0.74633333 0.7413     0.7329
 0.7271     0.7245    ]
```

Random forest

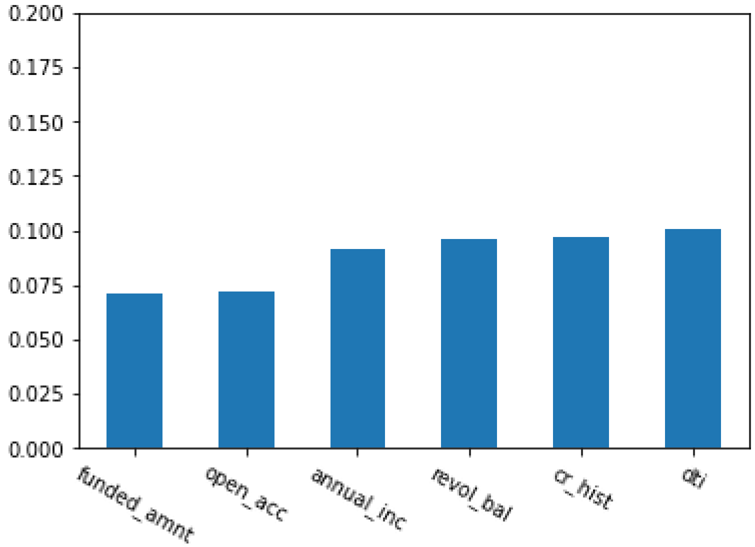
In [55]:
```
## Train and test a random forest classifier

random_forest = RandomForestClassifier()
cv_parameters = {'n_estimators': [100, 200, 300, 100],'max_depth': [80, 90, 100, 110]}
data_dict = prepare_data(feature_subset = your_features)
random_forest = fit_classification(random_forest,data_dict,cv_parameters =cv_parameters ,model_name ="Random Forest")
```

```
=========================================================
  Model: Random Forest
=========================================================
Fit time: 502.37 seconds
Optimal parameters:
{'max_depth': 110, 'n_estimators': 200}

Accuracy-maximizing threshold was: 0.57
Accuracy:  0.80385
              precision    recall  f1-score   support

  No default     0.8050    0.9976    0.8910     16079
     Default     0.4865    0.0092    0.0180      3921

    accuracy                         0.8038     20000
   macro avg     0.6458    0.5034    0.4545     20000
weighted avg     0.7426    0.8038    0.7199     20000
```




```
Similarity to LC grade ranking:  0.4798993594340665
Brier score: 0.14757652375
Were parameters on edge? : True
Score variations around CV search grid : 0.1458090318280265
[0.79973333 0.79896667 0.7995     0.79973333 0.7993     0.7994
 0.7999     0.79913333 0.79906667 0.7996     0.7992     0.799
 0.79966667 0.80013333 0.8        0.80003333]
```

In [56]:
```
## Plot top 6 most significant features
top_idx = list(np.argsort(random_forest['model'].feature_importances_)[-6:])
bplot = pd.Series(random_forest['model'].feature_importances_[top_idx])
xticks = selected_features[top_idx]
p2 = bplot.plot(kind='bar',rot=-30,ylim=(0,0.2))
p2.set_xticklabels(xticks)
plt.show()
```

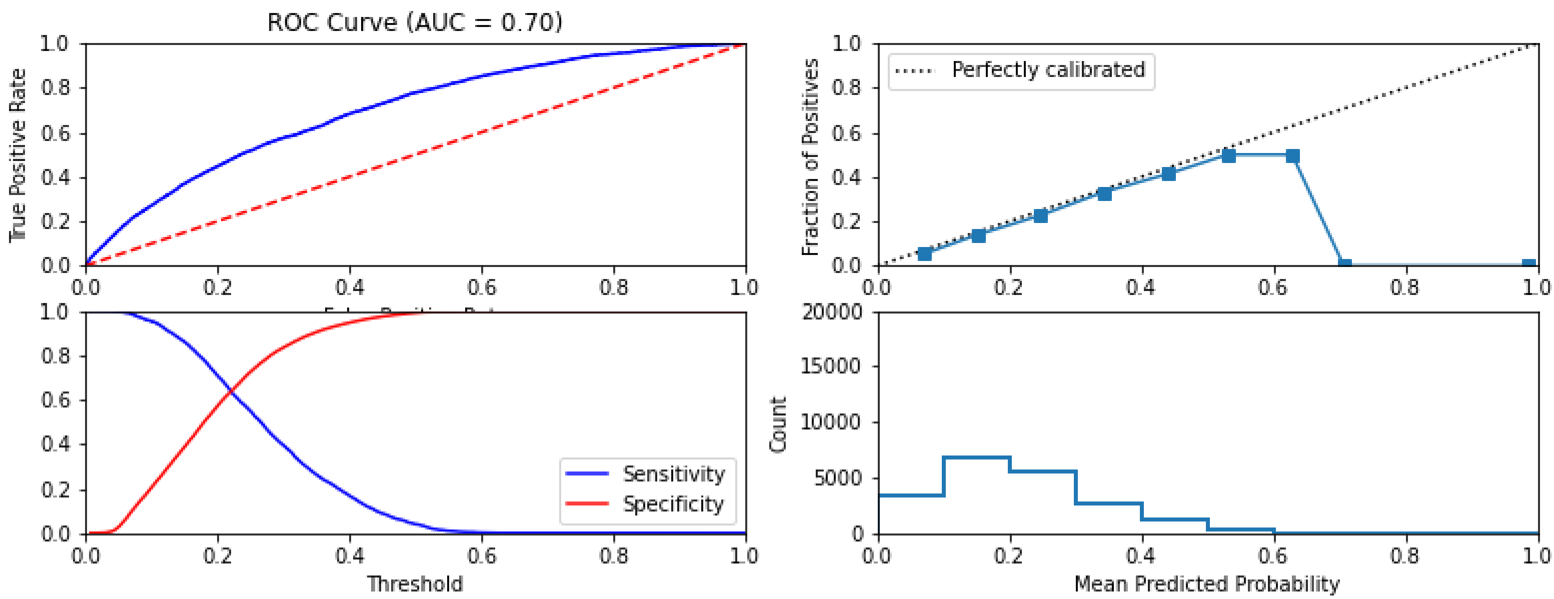


Multi-layer perceptron

In [ ]:
```
## Train and test a multi-layer perceptron classifier

mlp = MLPClassifier()
cv_parameters = {'hidden_layer_sizes': [20,40,60,80,100,120,140],'activation' : ['identity', 'logistic', 'tanh', 'relu'
data_dict = prepare_data(feature_subset = your_features)
mlp = fit_classification(mlp,data_dict,cv_parameters =cv_parameters ,model_name ="Multi-layer perceptron")
```

In [90]:
```
mlp_test= MLPClassifier(activation= 'identity', hidden_layer_sizes=100, learning_rate= 'constant', solver= 'adam')
data_dict = prepare_data(feature_subset = your_features)
mlp_1 = fit_classification(mlp_test,data_dict,model_name ="Multi-layer perceptron")
```

```
==========================================================
  Model: Multi-layer perceptron
==========================================================
Fit time: 23.87 seconds
Optimal parameters:
{}

Accuracy-maximizing threshold was: 0.5127940142210694
Accuracy:  0.8049
              precision    recall  f1-score   support

  No default     0.8081    0.9932    0.8911     16079
     Default     0.5401    0.0326    0.0616      3921

    accuracy                         0.8049     20000
   macro avg     0.6741    0.5129    0.4764     20000
weighted avg     0.7555    0.8049    0.7285     20000
```

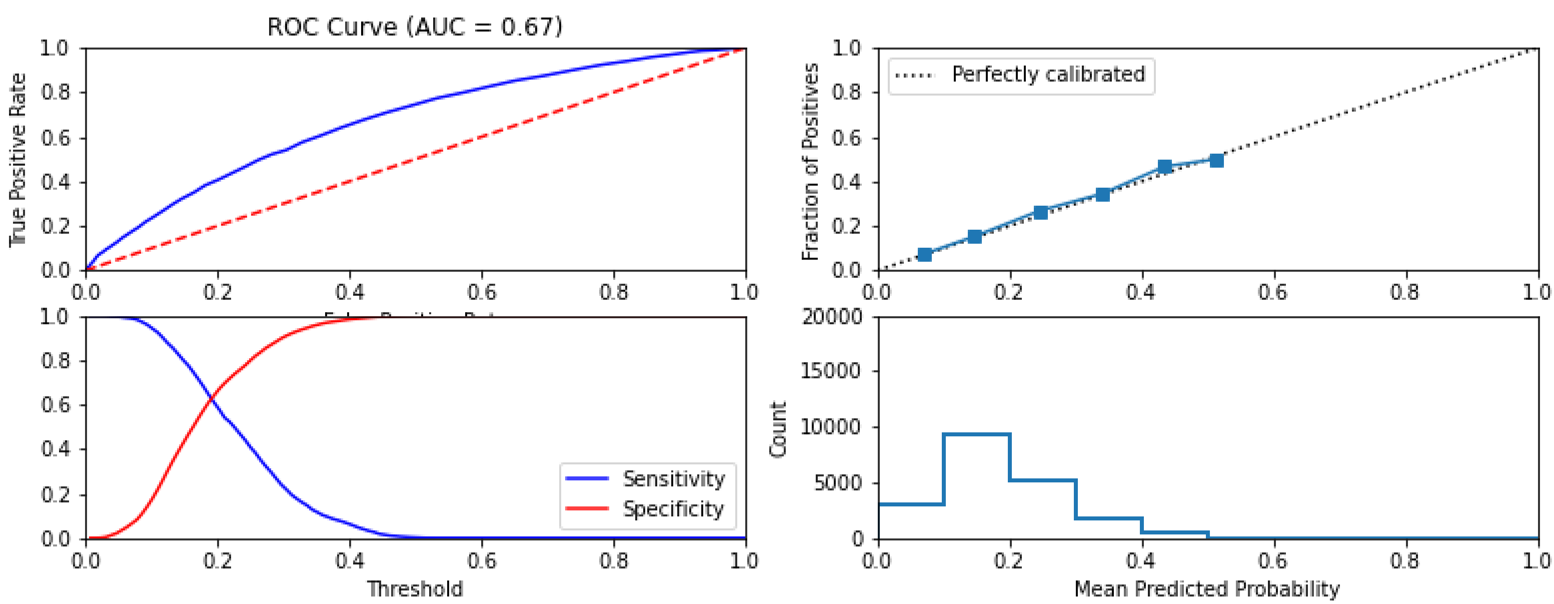



```
Similarity to LC grade ranking:  0.7546134776863364
Brier score: 0.14568840858428997
Were parameters on edge? : False
Score variations around CV search grid : 0.0
[0.80166667]
```

KNN

In [59]:
```
from sklearn.neighbors import KNeighborsClassifier
KNN = KNeighborsClassifier()
cv_parameters = {'n_neighbors':[1,2,4,8,16,32,64,128]}
data_dict = prepare_data(feature_subset = your_features)
knn = fit_classification(KNN,data_dict,cv_parameters =cv_parameters ,model_name ="KNN")
```

```
==========================================================
  Model: KNN
==========================================================
Fit time: 110.57 seconds
Optimal parameters:
{'n_neighbors': 128}

Accuracy-maximizing threshold was: 0.46875
Accuracy:  0.8037
              precision    recall  f1-score   support

  No default     0.8047    0.9980    0.8910     16079
     Default     0.4576    0.0069    0.0136      3921

    accuracy                         0.8037     20000
   macro avg     0.6312    0.5024    0.4523     20000
weighted avg     0.7367    0.8037    0.7190     20000
```



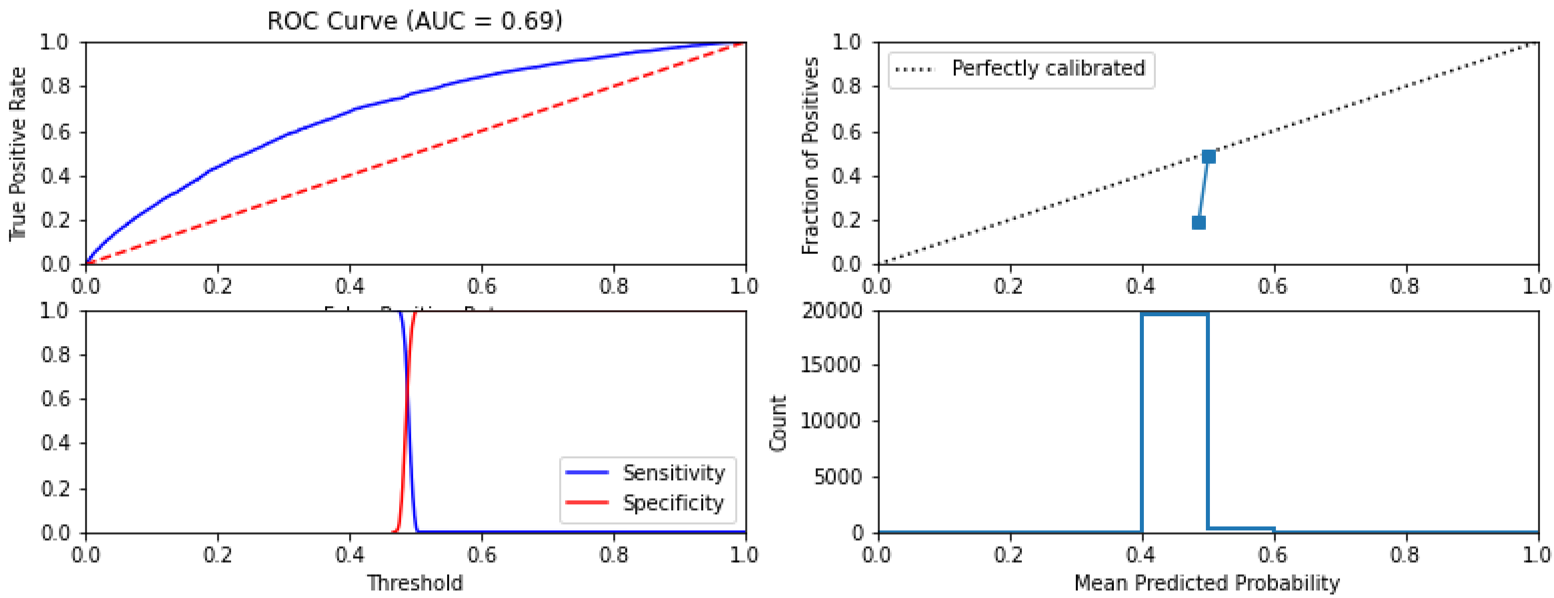
```
Similarity to LC grade ranking:  0.664142000201902
Brier score: 0.14863171997070312
Were parameters on edge? : True
Score variations around CV search grid : 12.600224336338327
[0.70126667 0.78073333 0.78576667 0.7928     0.79643333 0.80053333
 0.80233333 0.80236667]
```

AdaBoost

In [79]:

```
ada_clf=AdaBoostClassifier()
cv_parameters = {'n_estimators': [10,20,30,40,50,60,70,100]}
data_dict = prepare_data(feature_subset = your_features)
ada = fit_classification(ada_clf,
                         data_dict,
                         cv_parameters =cv_parameters ,
                         model_name ="AdaBoost classifier" )
```

```
==========================================================
  Model: AdaBoost classifier
==========================================================
Fit time: 62.84 seconds
Optimal parameters:
{'n_estimators': 30}

Accuracy-maximizing threshold was: 0.5025949258013708
Accuracy:  0.80345
              precision    recall  f1-score   support

  No default     0.8050    0.9971    0.8908     16079
     Default     0.4390    0.0092    0.0180      3921

    accuracy                         0.8034     20000
   macro avg     0.6220    0.5032    0.4544     20000
weighted avg     0.7332    0.8034    0.7197     20000
```




```
Similarity to LC grade ranking:  0.630887412101829
Brier score: 0.24093426326040435
Were parameters on edge? : False
Score variations around CV search grid : 0.10809462437115458
[0.80136667 0.80116667 0.80176667 0.8012     0.80163333 0.80113333
 0.8009     0.80096667]
```

Train and Test logistic regression model with features derived by LendingClub

In [50]:
```
random_state=[i for i in range(100)]
```

In [101…
```
## Find a LendingClub-defined feature and train a l1-regularized logistic regression model on data with only that featu
a_lendingclub_feature = 'grade'

data_dict = prepare_data(feature_subset = a_lendingclub_feature)
lc1_only_logistic = LogisticRegression(penalty='l1',solver = 'liblinear')
cv_parameters = {'C': [0.01, 0.1, 1.0, 10.0, 100.0],
                 'max_iter': [100, 500, 1000]}
performance=[]
for i in random_state:
    lc1_l1 = fit_classification(lc1_only_logistic,
                                data_dict,
```

```
                                    cv_parameters=cv_parameters,
                                    random_state = i,
                                    output_to_file = False,
                                    print_to_screen = False,
                                    model_name ="l_1 regularized logistic regression classifier with LC derived feature" )
    performance.append(lc1_l1['performance'])

avg_perf=np.average(performance)
sd=np.std(performance)
print("Average performance +- std of L1 Logistic regression with LC derived feature:",avg_perf+sd,avg_perf-sd)
```

```
Average performance +- std of L1 Logistic regression with LC derived feature: 0.7165781784417529 0.7165781784417529
```

In [102…

```
## train a l2-regularized logistic regression model on data with only that feature
lc2_only_logistic = LogisticRegression(penalty='l2')
cv_parameters = {'C': [0.01, 0.1, 1.0, 10.0, 100.0],
                 'max_iter': [100, 500, 1000]}
performance=[]
for i in random_state:
    lc2_l2 = fit_classification(lc2_only_logistic,
                                      data_dict,
                                      cv_parameters=cv_parameters,
                                      random_state=i,
                                      output_to_file=False,
                                      print_to_screen=False,
                                      model_name ="l_2 regularized logistic regression classifier with LC derived feat
    performance.append(lc2_l2['performance'])
avg_perf=np.average(performance)
sd=np.std(performance)
print("Average performance +- std of L2 Logistic regression with LC derived feature:",avg_perf+sd,avg_perf-sd)
```

```
Average performance +- std of L2 Logistic regression with LC derived feature: 0.7165781784417529 0.7165781784417529
```

Train and test all the models you have tried previously after removing features derived by LendingClub

In [103…

```
your_features_grade_removed = ['loan_amnt', 'funded_amnt', 'term', 'int_rate',
       'emp_length', 'home_ownership', 'annual_inc', 'verification_status',
       'issue_d', 'loan_status', 'purpose', 'dti', 'delinq_2yrs',
       'earliest_cr_line', 'open_acc', 'pub_rec', 'fico_range_high',
       'fico_range_low', 'revol_bal', 'revol_util', 'total_pymnt',
       'recoveries', 'last_pymnt_d', 'loan_length', 'term_num', 'ret_PESS',
```

```
       'ret_OPT', 'ret_INTa', 'ret_INTb', 'cr_hist', 'train']
# removed grade
data_dict = prepare_data(feature_subset = your_features_grade_removed)
gnb_performance=[]
l1_performance=[]
l2_performance=[]
decision_tree_performance=[]
rf_performance=[]
mlp_performance=[]
ada_performance=[]

for i in random_state:

    #NB
    gnb = GaussianNB(var_smoothing= 1)
    gnb_obj = fit_classification(gnb,
                                 data_dict,
                                 random_state=i,
                                 output_to_file=False,
                                 print_to_screen=False,
                                 model_name = 'Gaussian Naive Bayes')
    gnb_performance.append(gnb_obj['performance'])

    #l1 logistic
    l1_logistic = LogisticRegression(penalty='l1',solver = 'liblinear',C=0.1, max_iter=100)

    l1_logistic_obj = fit_classification(l1_logistic,
                                 data_dict,
                                 random_state=i,
                                 output_to_file=False,
                                 print_to_screen=False,
                                 model_name ="l_1 regularized logistic regression classifier" )
    l1_performance.append(l1_logistic_obj['performance'])

    #l2 logistic
    l2_logistic = LogisticRegression(penalty='l2',C=0.01, max_iter= 100)

    l2_logistic_obj = fit_classification(l2_logistic,
                                 data_dict,
                                 random_state=i,
                                 output_to_file=False,
                                 print_to_screen=False,
                                 model_name ="l_2 regularized logistic regression classifier" )
    l2_performance.append(l2_logistic_obj['performance'])
```

```
    #Decision tree
    decision_tree = DecisionTreeClassifier(max_depth= 1.0)
    decision_tree_obj= fit_classification(decision_tree,
                                       data_dict ,
                                       random_state=i,
                                       output_to_file=False,
                                       print_to_screen=False,
                                       model_name ="Decision Tree" )
    decision_tree_performance.append(decision_tree_obj['performance'])

    #RandomForest
    random_forest = RandomForestClassifier(max_depth= 100, n_estimators= 200)
    random_forest_obj = fit_classification(random_forest,
                                        data_dict,
                                        random_state=i,
                                        output_to_file=False,
                                        print_to_screen=False,
                                        model_name ="Random Forest")
    rf_performance.append(random_forest_obj['performance'])

    #MLP
    mlp = MLPClassifier(activation='identity', hidden_layer_sizes=60, learning_rate='constant', solver= 'sgd')
    mlp_obj = fit_classification(mlp,
                              data_dict,
                              random_state=i,
                              output_to_file=False,
                              print_to_screen=False,
                              model_name ="Multi-layer perceptron")
    mlp_performance.append(mlp_obj['performance'])

    #Adaboost
    ada_clf=AdaBoostClassifier(n_estimators= 10)
    ada = fit_classification(ada_clf,
                          data_dict,
                          random_state=i,
                          output_to_file=False,
                          print_to_screen=False,
                          model_name ="AdaBoost classifier" )
    ada_performance.append(ada['performance'])

print("Average performance +- std of Gaussian NB classifier without LC derived feature:",np.average(gnb_performance)+np
print("Average performance +- std of L1 Logistic Regression without LC derived feature:",np.average(l1_performance)+np.
print("Average performance +- std of L2 Logistic Regression without LC derived feature:",np.average(l2_performance)+np.
print("Average performance +- std of Decision Tree classifier without LC derived feature:",np.average(decision_tree_per
print("Average performance +- std of Random classifier without LC derived feature:",np.average(rf_performance)+np.std(r
```

```
print("Average performance +- std of MLP classifier without LC derived feature:",np.average(mlp_performance)+np.std(mlp
print("Average performance +- std of AdaBoost classifier without LC derived feature:",np.average(ada_performance)+np.st
```

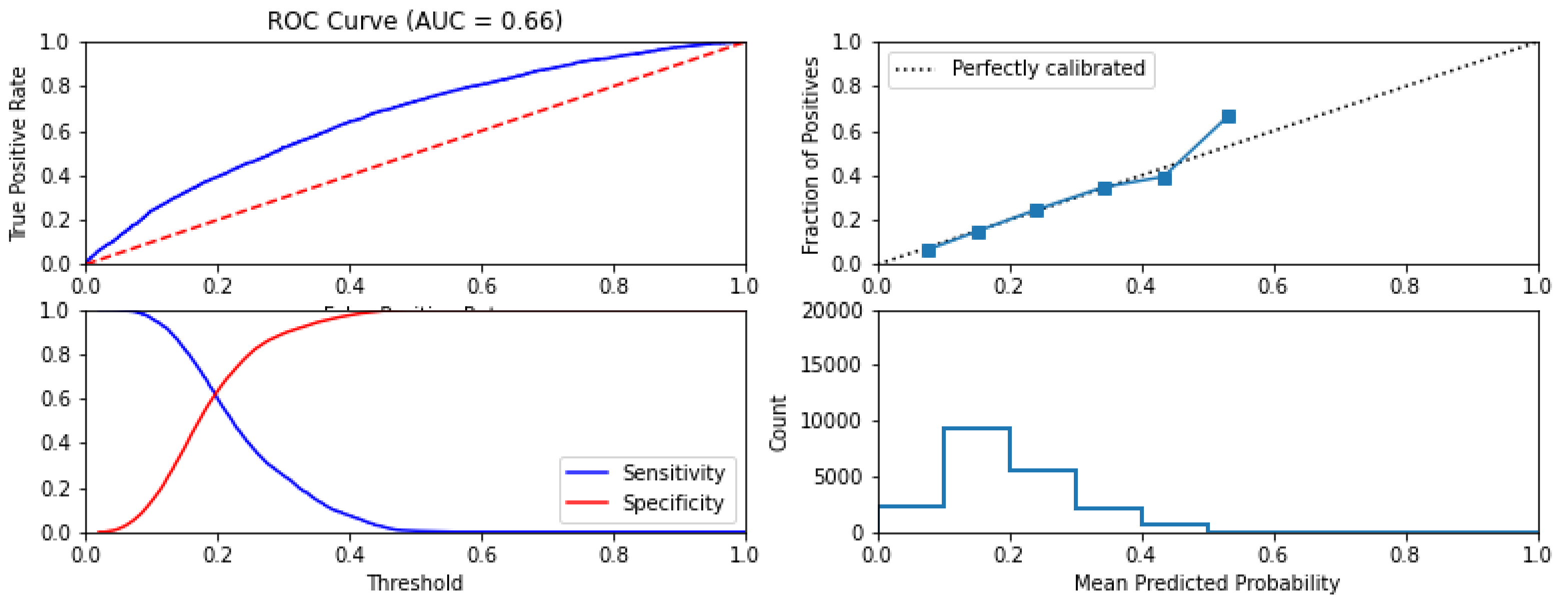
```
Average performance +- std of Gaussian NB classifier without LC derived feature: 0.7179318482238959 0.7179318482238957
Average performance +- std of L1 Logistic Regression without LC derived feature: 0.7200933105064655 0.7200551986529892
Average performance +- std of L2 Logistic Regression without LC derived feature: 0.7187308907190054 0.7187308907190048
Average performance +- std of Decision Tree classifier without LC derived feature: 0.7165781784417529 0.716578178441752
9
Average performance +- std of Random classifier without LC derived feature: 0.7199138126769689 0.7179140588273736
Average performance +- std of MLP classifier without LC derived feature: 0.7210555402547913 0.7178019790772292
Average performance +- std of AdaBoost classifier without LC derived feature: 0.7166980148598092 0.716698014859809
```

YOURMODEL

In [106…

```
l1_logistic = LogisticRegression(penalty='l1',solver = 'liblinear',C=0.1, max_iter=100)
data_dict = prepare_data(feature_subset = your_features_grade_removed)
l1_logistic_reg = fit_classification(l1_logistic,
                         data_dict,
                         model_name ="L1 Logistic Regression" )
```

```
=========================================================
  Model: L1 Logistic Regression
=========================================================
Fit time: 1.37 seconds
Optimal parameters:
{}

Accuracy-maximizing threshold was: 0.47391067214243765
Accuracy:  0.80415
              precision    recall  f1-score   support

  No default     0.8051    0.9980    0.8912     16079
     Default     0.5294    0.0092    0.0180      3921

    accuracy                         0.8042     20000
   macro avg     0.6672    0.5036    0.4546     20000
weighted avg     0.7510    0.8042    0.7200     20000
```



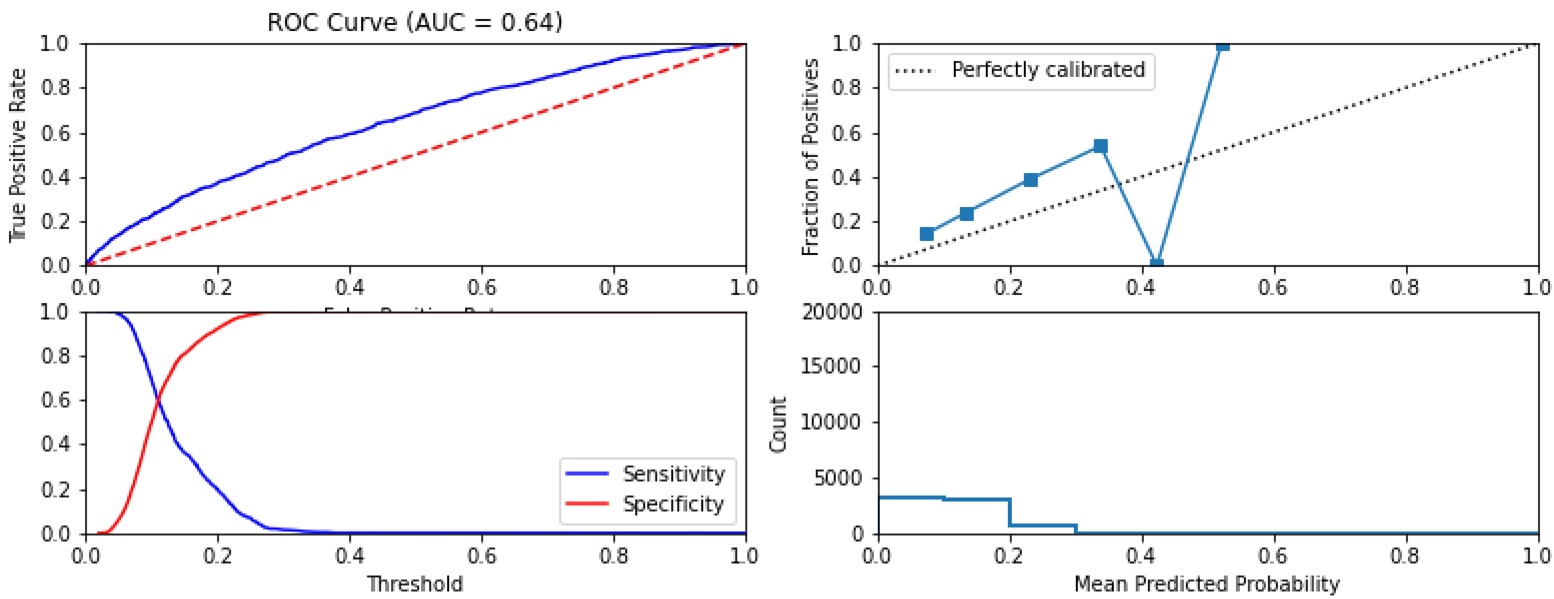

```
Similarity to LC grade ranking:  0.46621070267103637
Brier score: 0.14932006726304217
Were parameters on edge? : False
Score variations around CV search grid : 0.0
[0.80246667]
```

Time stability test of YOURMODEL

In [108…

```python
start_date_train = datetime.date(2010, 1, 1)
end_date_train = datetime.date(2010, 12, 31)
start_date_test = datetime.date(2018, 1, 1)
end_date_test = datetime.date(2018, 12, 1)

data_dict_test = prepare_data(date_range_train = (start_date_train, end_date_train),
                         date_range_test = (start_date_test, end_date_test),
                         n_samples_train = 7000, n_samples_test = 7000, feature_subset = your_features)

## Train and test YOURMODEL using this data
l1_logistic = LogisticRegression(penalty='l1',solver = 'liblinear',C=0.1, max_iter=100)
l1_2010 = fit_classification(l1_logistic,
                         data_dict_test,
                         model_name ="L1 logistic regression on 2010 data" )
```

```
=========================================================
  Model: L1 logistic regression on 2010 data
=========================================================
Fit time: 0.09 seconds
Optimal parameters:
{}

Accuracy-maximizing threshold was: 1
Accuracy:  0.7868571428571428
              precision    recall  f1-score   support

  No default     0.7869    1.0000    0.8807      5508
     Default     0.0000    0.0000    0.0000      1492

    accuracy                         0.7869      7000
   macro avg     0.3934    0.5000    0.4404      7000
weighted avg     0.6191    0.7869    0.6930      7000
```



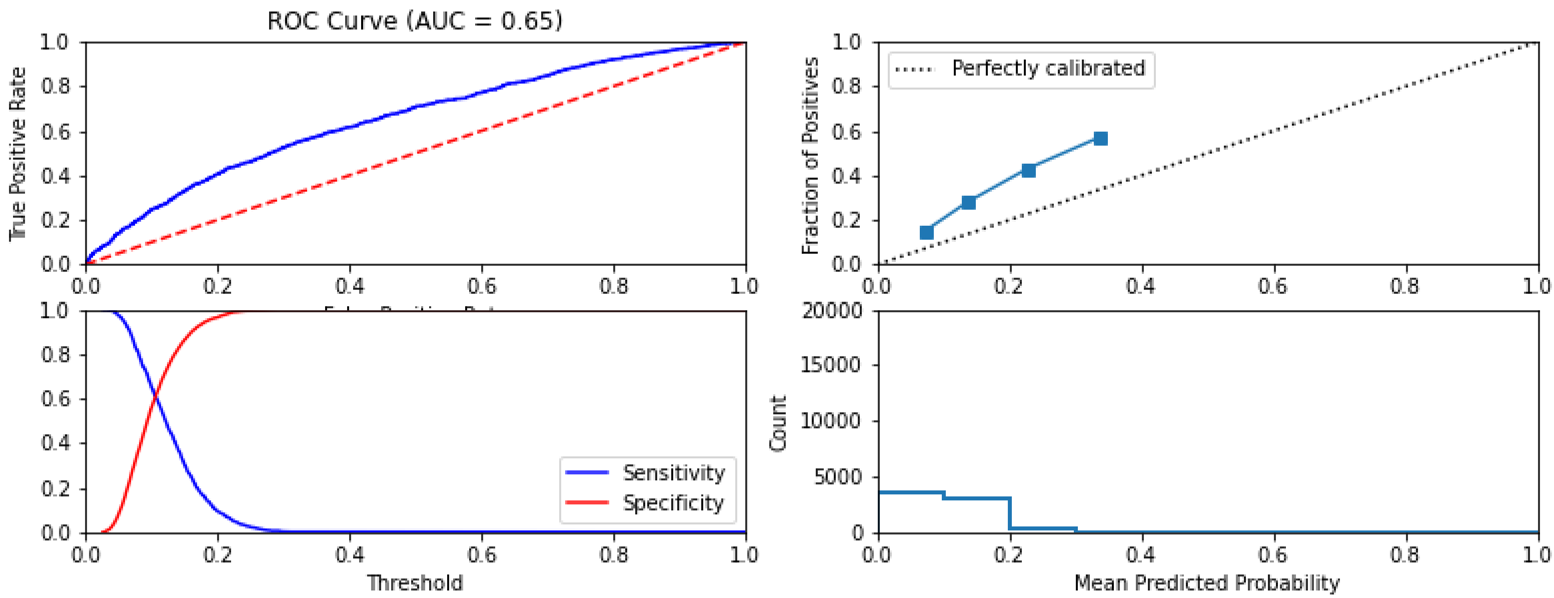

```
Similarity to LC grade ranking:  0.5633467949085471
Brier score: 0.17015676757739973
Were parameters on edge? : False
Score variations around CV search grid : 0.0
[0.88557143]
```

In [109…

```
start_date_train = datetime.date(2017, 1, 1)
end_date_train = datetime.date(2017, 12, 1)
```

```
start_date_test = datetime.date(2018, 1, 1)
end_date_test = datetime.date(2018, 12, 1)

data_dict_test = prepare_data(date_range_train = (start_date_train, end_date_train),
                         date_range_test = (start_date_test, end_date_test),
                         n_samples_train = 9000, n_samples_test = 7000, feature_subset = your_features)

## Train and test YOURMODEL using this data
l1_logistic = LogisticRegression(penalty='l1',solver = 'liblinear',C=0.1, max_iter=100)
l1_2017 = fit_classification(l1_logistic,
                         data_dict_test,
                         model_name ="L1 logistic regression on 2017 data" )
```

```
=========================================================
  Model: L1 logistic regression on 2017 data
=========================================================
Fit time: 0.31 seconds
Optimal parameters:
{}

Accuracy-maximizing threshold was: 1
Accuracy:  0.7792857142857142
              precision    recall  f1-score   support

  No default     0.7793    1.0000    0.8760      5455
     Default     0.0000    0.0000    0.0000      1545

    accuracy                         0.7793      7000
   macro avg     0.3896    0.5000    0.4380      7000
weighted avg     0.6073    0.7793    0.6826      7000
```




```
Similarity to LC grade ranking:  0.4915290697791847
Brier score: 0.17798739502487057
Were parameters on edge? : False
Score variations around CV search grid : 0.0
[0.885]
```

Test regression models

In [20]:
```python
def fit_regression(model, data_dict,
                      cv_parameters = {},
                      separate = False,
                      model_name = None,
                      random_state = default_seed,
                      output_to_file = True,
                      print_to_screen = True):
    '''
    This function will fit a regression model to data and print various evaluation
    measures. It expects the following parameters
      - model: an sklearn model object
      - data_dict: the dictionary containing both training and testing data;
                   returned by the prepare_data function
      - separate: a Boolean variable indicating whether we fit models for
                  defaulted and non-defaulted loans separately
      - cv_parameters: a dictionary of parameters that should be optimized
```

```
                    over using cross-validation. Specifically, each named
                    entry in the dictionary should correspond to a parameter,
                    and each element should be a list containing the values
                    to optimize over
      - model_name: the name of the model being fit, for printouts
      - random_state: the random seed to use
      - output_to_file: if the results will be saved to the output file
      - print_to_screen: if the results will be printed on screen

    This function returns a dictionary FOR EACH RETURN DEFINITION with the following entries
      - model: the best fitted model
      - predicted_return: prediction result based on the test set
      - predicted_regular_return: prediction result for non-defaulted loans (valid if separate == True)
      - predicted_default_return: prediction result for defaulted loans (valid if separate == True)
      - r2_scores: the testing r2_score(s) for the best fitted model
    '''

    np.random.seed(random_state)

    # --------------------------
    #   Step 1 - Load the data
    # --------------------------

    col_list = ['ret_PESS', 'ret_OPT', 'ret_INTa', 'ret_INTb']

    X_train = data_dict['X_train']
    filter_train = data_dict['train_set']

    X_test = data_dict['X_test']
    filter_test = data_dict['test_set']
    out = {}

    for ret_col in col_list:

        y_train = data.loc[filter_train, ret_col].to_numpy()
        y_test = data.loc[filter_test, ret_col].to_numpy()

        # --------------------------
        #   Step 2 - Fit the model
        # --------------------------

        if separate:
            outcome_train = data.loc[filter_train, 'outcome']
            outcome_test = data.loc[filter_test, 'outcome']
```

```python
    # Train two separate regressors for defaulted and non-defaulted loans
    X_train_0 = X_train[outcome_train == False]
    y_train_0 = y_train[outcome_train == False]
    X_test_0 = X_test[outcome_test == False]
    y_test_0 = y_test[outcome_test == False]

    X_train_1 = X_train[outcome_train == True]
    y_train_1 = y_train[outcome_train == True]
    X_test_1 = X_test[outcome_test == True]
    y_test_1 = y_test[outcome_test == True]

    cv_model_0 = GridSearchCV(model, cv_parameters, scoring='r2')
    cv_model_1 = GridSearchCV(model, cv_parameters, scoring='r2')

    start_time = time.time()
    cv_model_0.fit(X_train_0, y_train_0)
    cv_model_1.fit(X_train_1, y_train_1)
    end_time = time.time()

    best_model_0 = cv_model_0.best_estimator_
    best_model_1 = cv_model_1.best_estimator_

    if print_to_screen:

        if model_name != None:
            print("=========================================================")
            print("  Model: " + model_name + "  Return column: " + ret_col)
            print("=========================================================")

        print("Fit time: " + str(round(end_time - start_time, 2)) + " seconds")
        print("Optimal parameters:")
        print("model_0:",cv_model_0.best_params_, "model_1",cv_model_1.best_params_)

    predicted_regular_return = best_model_0.predict(X_test)
    predicted_default_return = best_model_1.predict(X_test)

    if print_to_screen:
        print("")
        print("Testing r2 scores:")
    # Here we use different testing set to report the performance
    test_scores = {'model_0':r2_score(y_test_0,best_model_0.predict(X_test_0)),
                   'model_1':r2_score(y_test_1,best_model_1.predict(X_test_1))}
    if print_to_screen:
        print("model_0:", test_scores['model_0'])
        print("model_1:", test_scores['model_1'])
```

```
            cv_objects = {'model_0':cv_model_0, 'model_1':cv_model_1}
            out[ret_col] = { 'model_0':best_model_0, 'model_1':best_model_1, 'predicted_regular_return':predicted_regul
                        'predicted_default_return':predicted_default_return,'r2_scores':test_scores }

        else:
            cv_model = GridSearchCV(model, cv_parameters, scoring='r2')

            start_time = time.time()
            cv_model.fit(X_train, y_train)
            end_time = time.time()

            best_model = cv_model.best_estimator_

            if print_to_screen:
                if model_name != None:
                    print("=========================================================")
                    print("  Model: " + model_name + "  Return column: " + ret_col)
                    print("=========================================================")

                print("Fit time: " + str(round(end_time - start_time, 2)) + " seconds")
                print("Optimal parameters:")
                print(cv_model.best_params_)

            predicted_return = best_model.predict(X_test)
            test_scores = {'model':r2_score(y_test,predicted_return)}
            if print_to_screen:
                print("")
                print("Testing r2 score:", test_scores['model'])

            cv_objects = {'model':cv_model}
            out[ret_col] = {'model':best_model, 'predicted_return':predicted_return, 'r2_scores':r2_score(y_test,predic

        # Output the results to a file
        if output_to_file:
            for i in cv_objects:
                # Check whether any of the CV parameters are on the edge of
                # the search space
                opt_params_on_edge = find_opt_params_on_edge(cv_objects[i])
                dump_to_output(model_name + "::" + ret_col + "::search_on_edge", opt_params_on_edge)
                if print_to_screen:
                    print("Were parameters on edge (" + i + ") : " + str(opt_params_on_edge))

                # Find out how different the scores are for the different values
                # tested for by cross-validation. If they're not too different, then
```

```
                # even if the parameters are off the edge of the search grid, we should
                # be ok
                score_variation = find_score_variation(cv_objects[i])
                dump_to_output(model_name + "::" + ret_col + "::score_variation", score_variation)
                if print_to_screen:
                    print("Score variations around CV search grid (" + i + ") : " + str(score_variation))

                # Print out all the scores
                dump_to_output(model_name + "::all_cv_scores", str(cv_objects[i].cv_results_['mean_test_score']))
                if print_to_screen:
                    print("All test scores : " + str(cv_objects[i].cv_results_['mean_test_score']) )

                # Dump the AUC to file
                dump_to_output( model_name + "::" + ret_col + "::r2", test_scores[i] )

    return out
```

l_1 regularized linear regression

In [30]:
```
## First, trying l1 regularized linear regression with hyper-parameters
reg_lasso = linear_model.Lasso()
cv_parameters = {'alpha': [0.01, 0.1, 1, 10]}
data_dict = prepare_data(feature_subset = your_features)
reg_lasso = fit_regression(reg_lasso ,data_dict,
                           cv_parameters = cv_parameters ,
                           model_name = 'l1 regularized linear regression',output_to_file=False )
```

```
============================================================
  Model: l1 regularized linear regression  Return column: ret_PESS
============================================================
Fit time: 0.4 seconds
Optimal parameters:
{'alpha': 0.01}

Testing r2 score: 0.07083962426978796
============================================================
  Model: l1 regularized linear regression  Return column: ret_OPT
============================================================
Fit time: 0.51 seconds
Optimal parameters:
{'alpha': 0.01}

Testing r2 score: 0.009382578195717528
============================================================
  Model: l1 regularized linear regression  Return column: ret_INTa
============================================================
Fit time: 0.41 seconds
Optimal parameters:
{'alpha': 0.01}

Testing r2 score: 0.0032362051697140126
============================================================
  Model: l1 regularized linear regression  Return column: ret_INTb
============================================================
Fit time: 0.52 seconds
Optimal parameters:
{'alpha': 0.01}

Testing r2 score: 0.015076960889027768
```

l_2 regularized linear regressor

In [31]:

```
## trying l2 regularized linear regression with hyper-parameters
reg_ridge = linear_model.Ridge()
cv_parameters = {'alpha': [0.01, 0.1, 1, 10]}
data_dict = prepare_data(feature_subset = your_features)
reg_ridge = fit_regression(reg_ridge ,data_dict,
                           cv_parameters = cv_parameters ,
                           model_name = 'l2 regularized linear regression',output_to_file=False )
```

```
==========================================================
  Model: l2 regularized linear regression  Return column: ret_PESS
==========================================================
Fit time: 0.29 seconds
Optimal parameters:
{'alpha': 0.1}

Testing r2 score: 0.09842640063162755
==========================================================
  Model: l2 regularized linear regression  Return column: ret_OPT
==========================================================
Fit time: 0.3 seconds
Optimal parameters:
{'alpha': 10}

Testing r2 score: 0.016045031052308478
==========================================================
  Model: l2 regularized linear regression  Return column: ret_INTa
==========================================================
Fit time: 0.31 seconds
Optimal parameters:
{'alpha': 1}

Testing r2 score: 0.034345931300323596
==========================================================
  Model: l2 regularized linear regression  Return column: ret_INTb
==========================================================
Fit time: 0.29 seconds
Optimal parameters:
{'alpha': 1}

Testing r2 score: 0.031026826406505226
```

Multi-layer perceptron regressor

In [85]:

```
## trying multi-layer perceptron regression with hyper-parameters

mlp_reg = MLPRegressor()
cv_parameters = {'activation':['identity','relu','logistic'],'alpha':[1.e-06, 1.e-05, 1.e-04, 1.e-03, 1.e-02]}
data_dict = prepare_data(feature_subset = your_features)
mlp_reg = fit_regression(mlp_reg, data_dict, cv_parameters=cv_parameters, model_name = "Multi-layer Perceptron Regressi
```

```
=========================================================
  Model: Multi-layer Perceptron Regression  Return column: ret_PESS
=========================================================
Fit time: 23802.14 seconds
Optimal parameters:
{'activation': 'logistic', 'alpha': 0.0001}

Testing r2 score: 0.09597736962079473
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -416.43894132713825
All test scores : [-0.07515327 -0.08168165 -0.07364349 -0.05760354 -0.09539033 -0.14602688
 -0.20395093 -0.19254868 -0.14749664 -0.14351257 -0.04086383 -0.09434788
 -0.03949178 -0.10674466 -0.04662987]
=========================================================
  Model: Multi-layer Perceptron Regression  Return column: ret_OPT
=========================================================
Fit time: 10116.86 seconds
Optimal parameters:
{'activation': 'identity', 'alpha': 0.01}

Testing r2 score: 0.01644052178309685
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -172.172692251619
All test scores : [-0.12108852 -0.15310054 -0.13407921 -0.1276937  -0.10883138 -0.25504366
 -0.29620931 -0.27475103 -0.28104524 -0.26389253 -0.12913617 -0.12796423
 -0.12812705 -0.15446004 -0.12355368]
=========================================================
  Model: Multi-layer Perceptron Regression  Return column: ret_INTa
=========================================================
Fit time: 299.99 seconds
Optimal parameters:
{'activation': 'logistic', 'alpha': 1e-06}

Testing r2 score: 0.030856705663676154
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -1363.567310491261
All test scores : [-0.07135528 -0.03133361 -0.01782331 -0.03066222 -0.01912812 -0.05522552
 -0.06448616 -0.07443936 -0.09031523 -0.05755573 -0.0061709  -0.01444065
 -0.01649894 -0.04593296 -0.02172011]
=========================================================
  Model: Multi-layer Perceptron Regression  Return column: ret_INTb
=========================================================
Fit time: 1121.33 seconds
Optimal parameters:
{'activation': 'logistic', 'alpha': 0.001}
```

```
Testing r2 score: 0.028657412990256703
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 2270.899592878945
All test scores : [-0.00365445  0.00109074 -0.01498758 -0.01866798 -0.00279117 -0.14756043
 -0.15985659 -0.15231361 -0.13758031 -0.09867757  0.00668005  0.00305858
  0.00674874  0.00736361  0.00216942]
```

Random forest regressor

In [86]:
```
## trying random forest regression with hyper-parameters
rf = RandomForestRegressor()
cv_parameters = {'max_depth': [80, 90, 100, 110],'max_features': [2, 3],'min_samples_leaf': [3, 4, 5],'min_samples_spli
data_dict = prepare_data(feature_subset = your_features)
reg_rf = fit_regression(rf, data_dict=data_dict, cv_parameters=cv_parameters,model_name = "Random Forest Regression")
```

```
==========================================================
  Model: Random Forest Regression  Return column: ret_PESS
==========================================================
Fit time: 2736.61 seconds
Optimal parameters:
{'max_depth': 80, 'max_features': 3, 'min_samples_leaf': 3, 'min_samples_split': 10, 'n_estimators': 300}

Testing r2 score: 0.10537891643691144
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -24.183237018039183
All test scores : [-0.08059741 -0.07862441 -0.07926844 -0.0796374  -0.08062172 -0.07816915
 -0.08041898 -0.07903626 -0.07911791 -0.08274852 -0.08218663 -0.08197827
 -0.08285717 -0.07808129 -0.08127953 -0.08316301 -0.0810781  -0.08094868
 -0.0835796  -0.08178095 -0.081344   -0.08297552 -0.08321686 -0.08254985
 -0.08379863 -0.08158935 -0.0825657  -0.07089544 -0.07072853 -0.06859701
 -0.07164524 -0.06997048 -0.06824871 -0.07114041 -0.06968568 -0.06930987
 -0.07261955 -0.07025543 -0.07011256 -0.07130601 -0.07135942 -0.07133094
 -0.07205519 -0.07088328 -0.07024227 -0.07129255 -0.07129044 -0.0720792
 -0.07152432 -0.07043435 -0.07065723 -0.07092008 -0.0714376  -0.07189246
 -0.07951639 -0.07919281 -0.07907659 -0.07901458 -0.07913024 -0.07971236
 -0.08059348 -0.07905927 -0.07865089 -0.08258569 -0.08051464 -0.08075094
 -0.08241744 -0.08219443 -0.0797493  -0.08321278 -0.08114558 -0.08042984
 -0.08294765 -0.08328013 -0.08304815 -0.08458819 -0.08236397 -0.0821982
 -0.08423996 -0.08301505 -0.08233575 -0.06997725 -0.07073584 -0.06905497
 -0.07074511 -0.07110949 -0.06956273 -0.06947835 -0.07132101 -0.06913878
 -0.07200706 -0.0694818  -0.06975066 -0.07145854 -0.07160602 -0.07059559
 -0.0717991  -0.07062314 -0.07040932 -0.07222663 -0.07047328 -0.0715021
 -0.07104903 -0.07126403 -0.07023434 -0.07230909 -0.07036304 -0.07060125
 -0.07900587 -0.07969014 -0.07962159 -0.07947889 -0.07959955 -0.07855537
 -0.07974144 -0.07956193 -0.07964877 -0.08310217 -0.08135932 -0.08097506
 -0.08297332 -0.08097802 -0.0802116  -0.08126099 -0.08064974 -0.08116432
 -0.08305517 -0.08323588 -0.08214771 -0.08303399 -0.0822996  -0.08184862
 -0.08475345 -0.08130816 -0.08295914 -0.06869681 -0.07079587 -0.06926392
 -0.07332523 -0.07018781 -0.07044717 -0.07199994 -0.07031904 -0.06873143
 -0.07353265 -0.07015391 -0.07067149 -0.0726603  -0.07099607 -0.06955808
 -0.07118443 -0.06970833 -0.07047501 -0.07170549 -0.07068566 -0.07010742
 -0.07262704 -0.07192258 -0.07080592 -0.07249748 -0.07132497 -0.07091159
 -0.08149868 -0.07912366 -0.07801845 -0.07947646 -0.07935733 -0.07881763
 -0.0805391  -0.0794042  -0.08040725 -0.08356299 -0.08139773 -0.08136886
 -0.08042231 -0.08010255 -0.08149493 -0.08256668 -0.08222852 -0.08021597
 -0.08214431 -0.08405986 -0.0829795  -0.08287927 -0.08370874 -0.08185799
 -0.08306404 -0.08191017 -0.08255015 -0.071985   -0.0702253  -0.06895413
 -0.07229775 -0.07077048 -0.06891563 -0.07087887 -0.070553   -0.0695916
 -0.07149319 -0.06992996 -0.07044635 -0.0717652  -0.07165534 -0.0702967
 -0.07133566 -0.07002781 -0.07052441 -0.07247578 -0.07068785 -0.07088451
```

```
 -0.07412351 -0.07016189 -0.07138187 -0.07319948 -0.07023493 -0.07019932]
==========================================================
  Model: Random Forest Regression  Return column: ret_OPT
==========================================================
Fit time: 2894.81 seconds
Optimal parameters:
{'max_depth': 110, 'max_features': 2, 'min_samples_leaf': 5, 'min_samples_split': 12, 'n_estimators': 200}

Testing r2 score: 0.01960117990415644
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -6.67114673718371
All test scores : [-0.12161966 -0.12065659 -0.12055403 -0.12263064 -0.11989298 -0.12032211
 -0.12124365 -0.12133329 -0.11982503 -0.12059456 -0.11993492 -0.11845221
 -0.12183641 -0.11953847 -0.11932025 -0.12092854 -0.11884152 -0.11905652
 -0.12034125 -0.11903127 -0.11914781 -0.12039511 -0.11914728 -0.11887686
 -0.12181682 -0.11839457 -0.11876753 -0.12477299 -0.12144453 -0.121661
 -0.12413913 -0.12222364 -0.12097166 -0.12255326 -0.12079928 -0.12080055
 -0.12102631 -0.11968884 -0.11907192 -0.12346345 -0.12092467 -0.11990817
 -0.12147683 -0.11925661 -0.11905217 -0.11964593 -0.11959753 -0.11854459
 -0.12036794 -0.11896403 -0.11810135 -0.12186351 -0.11922324 -0.11826605
 -0.12265209 -0.12126242 -0.11958734 -0.12085666 -0.12012177 -0.12017521
 -0.12238616 -0.12124624 -0.12041433 -0.12115124 -0.11881873 -0.11999098
 -0.1212484  -0.12001605 -0.11943796 -0.12162411 -0.11948972 -0.11929736
 -0.1211921  -0.11916289 -0.11980169 -0.12075697 -0.11887221 -0.11878377
 -0.11905186 -0.11879487 -0.1189494  -0.12305163 -0.12221224 -0.12214157
 -0.12502784 -0.12194415 -0.12134447 -0.12263503 -0.12241043 -0.12140049
 -0.12229872 -0.12011924 -0.11968092 -0.1227801  -0.12011709 -0.11874454
 -0.12069038 -0.11995533 -0.12019924 -0.12120939 -0.11927647 -0.11868712
 -0.12103828 -0.12071608 -0.11819868 -0.12130482 -0.11939027 -0.11890898
 -0.1232205  -0.11932877 -0.12032036 -0.12137597 -0.12066043 -0.1199713
 -0.12249411 -0.12080162 -0.11928842 -0.12123474 -0.11949012 -0.11911587
 -0.11947864 -0.12025693 -0.1203033  -0.11883152 -0.11968453 -0.11879777
 -0.12023304 -0.1191303  -0.11826808 -0.1203885  -0.12005482 -0.11874661
 -0.11994715 -0.11925935 -0.11889404 -0.12222602 -0.12237645 -0.12146028
 -0.1254518  -0.12136594 -0.12230785 -0.12318438 -0.12259664 -0.12054999
 -0.12363063 -0.11996461 -0.12067843 -0.12217263 -0.11998968 -0.11934673
 -0.12009872 -0.1203184  -0.1190539  -0.12056246 -0.119103   -0.11912803
 -0.12099832 -0.12059921 -0.11886676 -0.12027052 -0.11934211 -0.11918536
 -0.12279292 -0.12137338 -0.12089796 -0.12213895 -0.12103571 -0.12034611
 -0.12206962 -0.12135373 -0.11922101 -0.12050707 -0.12033517 -0.11882682
 -0.11935545 -0.11780057 -0.11899709 -0.12034844 -0.11891983 -0.11871572
 -0.12101131 -0.11888844 -0.11800759 -0.11926131 -0.11948578 -0.11895672
 -0.11770241 -0.11760612 -0.11878215 -0.12413046 -0.12296151 -0.12140176
 -0.12529706 -0.12331443 -0.12102826 -0.12390969 -0.12226274 -0.12057095
 -0.12174389 -0.11973777 -0.12003201 -0.12324816 -0.1193553  -0.1200056
```

```
 -0.12222067 -0.1200592  -0.11907939 -0.1206782  -0.12021051 -0.11924615
 -0.12028489 -0.11942355 -0.11906151 -0.11990069 -0.11952796 -0.11867542]
==========================================================
  Model: Random Forest Regression  Return column: ret_INTa
==========================================================
Fit time: 2720.26 seconds
Optimal parameters:
{'max_depth': 100, 'max_features': 3, 'min_samples_leaf': 5, 'min_samples_split': 12, 'n_estimators': 300}

Testing r2 score: 0.04047990029498494
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 84.59618298615253
All test scores : [0.00209016 0.00361411 0.00423802 0.00220833 0.00392085 0.00364059
 0.00106818 0.00391756 0.00405956 0.0024371  0.00407713 0.00385986
 0.00288638 0.00382043 0.00427733 0.0028974  0.00343509 0.00460175
 0.00286382 0.00402545 0.00444434 0.00277365 0.00356477 0.00370788
 0.00282883 0.00420163 0.00437624 0.00119993 0.00362926 0.00464993
 0.00140804 0.00352786 0.004205   0.00164077 0.00365935 0.00556711
 0.00437219 0.0044129  0.00534812 0.00296066 0.00550373 0.00549898
 0.0044207  0.0042876  0.00593734 0.00401104 0.00507218 0.00568509
 0.00365863 0.00512212 0.00613081 0.0047629  0.00609881 0.00621579
 0.0015842  0.00292419 0.00297695 0.00276876 0.00291075 0.00354289
 0.00252352 0.00386822 0.00385283 0.00268023 0.00382658 0.00436122
 0.00327777 0.00409648 0.0044741  0.00234095 0.00337409 0.00497668
 0.0024535  0.00363759 0.00401879 0.00297593 0.00466794 0.00478275
 0.00354348 0.00366029 0.00460189 0.00212141 0.00404221 0.00459889
 0.00098539 0.00303767 0.00448093 0.00347068 0.00409294 0.00550868
 0.00502298 0.00422876 0.00487974 0.00333927 0.00550078 0.00597294
 0.00301083 0.00557944 0.00517577 0.00475091 0.00584744 0.00579546
 0.00475843 0.00627898 0.0061079  0.00486893 0.00508618 0.00615424
 0.00223292 0.00330201 0.00446662 0.00137502 0.00333628 0.00356547
 0.00286935 0.00367999 0.00496599 0.0037274  0.00378866 0.00401459
 0.00378058 0.00426138 0.00386079 0.00277946 0.00389957 0.00402118
 0.00307572 0.00331728 0.00486886 0.00340718 0.00379436 0.00494463
 0.00342363 0.00362148 0.00474761 0.00278472 0.00415654 0.00437628
 0.00186818 0.00354405 0.00419096 0.00378382 0.00371711 0.00385211
 0.00386021 0.0042558  0.00529264 0.00430273 0.00488292 0.00497995
 0.00331963 0.00536109 0.00489078 0.00404454 0.00606099 0.00596455
 0.00461697 0.0052826  0.00463011 0.00448022 0.00547926 0.00639707
 0.002498   0.00256838 0.00427435 0.00172958 0.00390669 0.00341946
 0.0020856  0.00383769 0.00394745 0.00149132 0.00266741 0.00436541
 0.00285791 0.00310764 0.00337892 0.00347777 0.00388957 0.00419606
 0.00335684 0.0043103  0.00468478 0.00368164 0.00378898 0.00442001
 0.00299766 0.00366013 0.00440782 0.00162446 0.00310786 0.00407178
 0.00326286 0.00347829 0.00473424 0.00272879 0.00470306 0.00469511
```

```
 0.00191897 0.00393674 0.00523896 0.00334032 0.00516514 0.00529885
 0.00381737 0.0048517  0.00539579 0.0041127  0.00554419 0.0058481
 0.00436661 0.00533171 0.00570788 0.00383422 0.00522099 0.00557131]
=========================================================
  Model: Random Forest Regression  Return column: ret_INTb
=========================================================
Fit time: 2218.73 seconds
Optimal parameters:
{'max_depth': 110, 'max_features': 3, 'min_samples_leaf': 5, 'min_samples_split': 8, 'n_estimators': 300}

Testing r2 score: 0.03769192670567301
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 27.21493435036017
All test scores : [0.01487036 0.01713299 0.01789269 0.01590795 0.01837359 0.01755803
 0.01623584 0.01772323 0.01860713 0.01716984 0.01846964 0.01859076
 0.0171538  0.01858799 0.01819881 0.01733812 0.01908317 0.01799384
 0.01752855 0.01846895 0.0184676  0.01846926 0.01867066 0.01823301
 0.01789039 0.01843506 0.018906   0.01515857 0.01758132 0.01766293
 0.01596561 0.01724715 0.01865448 0.01704606 0.01790024 0.01854839
 0.01678992 0.01819801 0.01875534 0.01681644 0.01853675 0.0188621
 0.0175845  0.01864413 0.01871159 0.01804426 0.01942042 0.02034419
 0.01770244 0.01888368 0.019082   0.01851637 0.01965636 0.01947678
 0.01504845 0.01744078 0.01807734 0.01510204 0.01716682 0.01722596
 0.01646647 0.01774584 0.01788751 0.01672649 0.01797316 0.01912042
 0.01714728 0.01736302 0.0185559  0.01800525 0.01818271 0.01821369
 0.01824932 0.01776187 0.01845275 0.01658221 0.01850768 0.01874052
 0.0172741  0.01891826 0.01864729 0.01504725 0.01748598 0.01842361
 0.01663609 0.01857206 0.0178324  0.01550946 0.01821192 0.01877087
 0.01697781 0.01828618 0.01875181 0.01741962 0.01835138 0.01801183
 0.01742538 0.01804129 0.01984623 0.01867726 0.01875689 0.01925359
 0.01798591 0.01965668 0.01951319 0.01668906 0.0189078  0.01975836
 0.01647623 0.01668906 0.01805304 0.01531363 0.01819255 0.01760451
 0.01545908 0.01798382 0.01842054 0.01649914 0.01852326 0.01860051
 0.01628457 0.01854905 0.01839091 0.01800171 0.01829878 0.01883651
 0.01715303 0.01797827 0.01897583 0.01674395 0.01855395 0.01880986
 0.01788796 0.01825415 0.01872531 0.01560971 0.01701011 0.01802741
 0.0165483  0.01797621 0.01784372 0.01647989 0.01776726 0.0176693
 0.01713334 0.01891144 0.01969003 0.01827089 0.01813903 0.01919383
 0.01704648 0.01959731 0.01914223 0.01763042 0.01895203 0.01956387
 0.01828087 0.01900993 0.01968786 0.01680559 0.01909181 0.0199583
 0.01505504 0.01798746 0.01721042 0.01618685 0.0178519  0.01790766
 0.01645537 0.01784428 0.01764384 0.01786488 0.01810342 0.0184941
 0.01722168 0.01849938 0.01900464 0.01815783 0.018279   0.01820673
 0.01726789 0.01862705 0.01922262 0.01883454 0.01797276 0.01881311
 0.0176565  0.01837623 0.01860375 0.01537632 0.01708845 0.01825422
```

```
0.01571592 0.01809243 0.01826066 0.01640755 0.01770535 0.01937959
0.01571647 0.01874637 0.01885823 0.01694282 0.01820733 0.01866714
0.01732482 0.01878626 0.01927951 0.01771539 0.01806154 0.0204305
0.0175626  0.0195014  0.01949395 0.0172396  0.01941673 0.02040544]
```

Test investment strategies

Now we test several investment strategies using the learning models above

In [27]:
```python
def test_investments(data_dict,
                        classifier = None,
                        regressor = None,
                        strategy = 'Random',
                        num_loans = 1000,
                        random_state = default_seed,
                        output_to_file = True):
    '''
    This function tests a variety of investment methodologies and their returns.
    It will run its tests on the loans defined by the test_set element of the data
    dictionary.

    It is currently able to test four strategies
      - random: invest in a random set of loans
      - default-based: score each loan by probability of default, and only invest
                in the "safest" loans (i.e., those with the lowest probabilities
                of default)
      - return-based: train a single regression model to predict the expected return
                   of loans in the past. Then, for loans we could invest in, simply
                   rank them by their expected returns and invest in that order.
      - default-& return-based: train two regression models to predict the expected return of
                  defaulted loans and non-defaulted loans in the training set. Then,
                  for each potential loan we could invest in, predict the probability
                  the loan will default, its return if it doesn't default and its
                  return if it does. Then, calculate a weighted combination of
                  the latter using the former to find a predicted return. Rank the
                  loans by this expected return, and invest in that order

    It expects the following parameters
      - data_dict: the dictionary containing both training and testing data;
                  returned by the prepare_data function
      - classifier: a fitted model object which is returned by the fit_classification function.
      - regressor: a fitted model object which is returned by the fit_regression function.
      - strategy: the name of the strategy; one of the three listed above
```

```
      - num_loans: the number of loans to be included in the test portfolio
      - num_samples: the number of random samples used to compute average return ()
      - random_state: the random seed to use when selecting a subset of rows
      - output_to_file: if the results will be saved to the output file

    The function returns a dictionary FOR EACH RETURN DEFINITION with the following entries
      - strategy: the name of the strategy
      - average return: the return of the strategy based on the testing set
      - test data: the updated Dataframe of testing data. Useful in the optimization section
    '''

    np.random.seed(random_state)

    # Retrieve the rows that were used to train and test  the
    # classification model
    train_set = data_dict['train_set']
    test_set = data_dict['test_set']

    col_list = ['ret_PESS', 'ret_OPT', 'ret_INTa', 'ret_INTb']

    # Create a dataframe for testing, including the score
    data_test = data.loc[test_set,:]
    out = {}

    for ret_col in col_list:

        if strategy == 'Random':
            # Randomize the order of the rows in the datframe
            data_test = data_test.sample(frac = 1).reset_index(drop = True)

            ## Select num_loans to invest in
            pf_test = data_test.iloc[:num_loans]

            ## Find the average return for these loans
            ret_test = pf_test[ret_col].mean()

            # Return
            out[ret_col] = {'strategy':strategy, 'average return':ret_test}

            # Dump the strategy performance to file
            if output_to_file:
                dump_to_output(strategy + "," + ret_col + "::average return", ret_test )

            continue
```

```
        elif strategy == 'Return-based':

            colname = 'predicted_return_' + ret_col

            data_test[colname] = regressor[ret_col]['predicted_return']

            # Sort the loans by predicted return
            data_test = data_test.sort_values(by=colname, ascending = False).reset_index(drop = True)

            ## Pick num_loans loans
            pf_test = data_test.iloc[:num_loans]

            ## Find their return
            ret_test = pf_test[ret_col].mean()

            # Return
            out[ret_col] = {'strategy':strategy, 'average return':ret_test, 'test data':data_test}

            # Dump the strategy performance to file
            if output_to_file:
                dump_to_output(strategy + "," + ret_col + "::average return", ret_test )

            continue

        # Get the predicted scores, if the strategy is not Random or just Regression
        try:
            y_pred_score = classifier['y_pred_probs']
        except:
            y_pred_score = classifier['y_pred_score']

        data_test['score'] = y_pred_score


        if strategy == 'Default-based':
            # Sort the test data by the score
            data_test = data_test.sort_values(by='score').reset_index(drop = True)

            ## Select num_loans to invest in
            pf_test = data_test.iloc[:num_loans]

            ## Find the average return for these loans
            ret_test = pf_test[ret_col].mean()

            # Return
            out[ret_col] = {'strategy':strategy, 'average return':ret_test}
```

```
            # Dump the strategy performance to file
            if output_to_file:
                dump_to_output(strategy + "," + ret_col + "::average return", ret_test )

            continue


        elif strategy == 'Default-return-based':

            # Load the predicted returns
            data_test['predicted_regular_return'] = regressor[ret_col]['predicted_regular_return']
            data_test['predicted_default_return'] = regressor[ret_col]['predicted_default_return']

            # Compute expectation
            colname = 'predicted_return_' + ret_col

            data_test[colname] = ( (1-data_test.score)*data_test.predicted_regular_return +
                                             data_test.score*data_test.predicted_default_return )

            # Sort the loans by predicted return
            data_test = data_test.sort_values(by=colname, ascending = False).reset_index(drop = True)

            ## Pick num_loans loans
            pf_test = data_test.iloc[:num_loans]

            ## Find their return
            ret_test = pf_test[ret_col].mean()

            # Return
            out[ret_col] = {'strategy':strategy, 'average return':ret_test, 'test data':data_test}

            # Dump the strategy performance to file
            if output_to_file:
                dump_to_output(strategy + "," + ret_col + "::average return", ret_test )

            continue

        else:
            return 'Not a valid strategy'

    return out
```

In [40]:

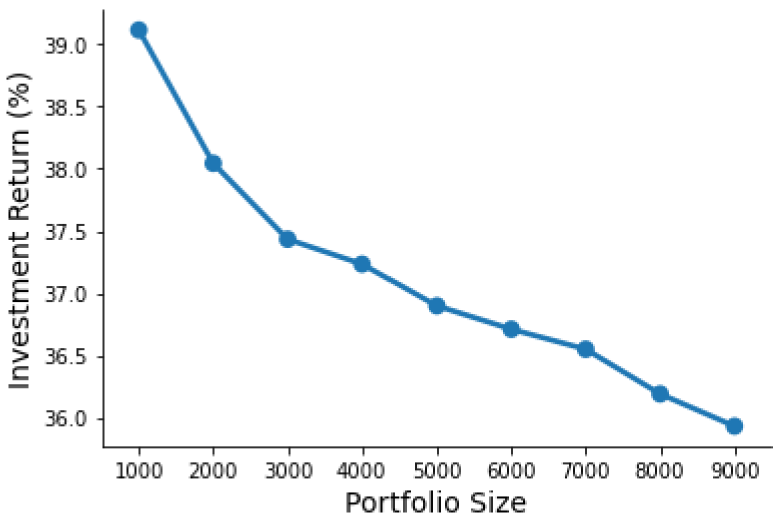
```
test_strategy = 'Default-return-based'
```

```
## For the Default-return-based strategy we need to fit a new regressor with separate = True
cv_parameters = {'alpha': [0.01, 0.1, 1, 10]}
reg_ridge_sep = linear_model.Ridge()
reg_separate = fit_regression(reg_ridge_sep ,data_dict,
                           cv_parameters = cv_parameters ,
                           model_name = 'l2 regularized linear regression separate',separate=True,output_to_file=False)

print('strategy:',test_strategy)
strat_defret = test_investments(data_dict=data_dict,classifier=ada,regressor=reg_separate,strategy=test_strategy,num_lo
                        output_to_file = True)

for ret_col in col_list:
    print(ret_col + ': ' + str(strat_defret[ret_col]['average return']))
```

```
==========================================================
  Model: l2 regularized linear regression separate  Return column: ret_PESS
==========================================================
Fit time: 0.34 seconds
Optimal parameters:
model_0: {'alpha': 0.1} model_1 {'alpha': 10}

Testing r2 scores:
model_0: 0.08535582472979653
model_1: 0.11908984553058921
==========================================================
  Model: l2 regularized linear regression separate  Return column: ret_OPT
==========================================================
Fit time: 0.34 seconds
Optimal parameters:
model_0: {'alpha': 10} model_1 {'alpha': 10}

Testing r2 scores:
model_0: 0.020848223834026514
model_1: 0.014813690920591926
==========================================================
  Model: l2 regularized linear regression separate  Return column: ret_INTa
==========================================================
Fit time: 0.33 seconds
Optimal parameters:
model_0: {'alpha': 10} model_1 {'alpha': 10}

Testing r2 scores:
model_0: 0.030475281304978452
model_1: 0.03671529376131899
==========================================================
  Model: l2 regularized linear regression separate  Return column: ret_INTb
==========================================================
Fit time: 0.34 seconds
Optimal parameters:
model_0: {'alpha': 10} model_1 {'alpha': 10}

Testing r2 scores:
model_0: 0.02778382760007747
model_1: 0.05177865546900684
strategy: Default-return-based
ret_PESS: 0.3084955671960841
ret_OPT: 1.4387481452548503
ret_INTa: 0.4128266687067464
ret_INTb: 1.2329303552812942
```

Sensitivity test of portfolio size

In [43]:
```
## Test the best-performing data-driven strategy on different portfolio sizes

result_sensitivity = []

## Vary the portfolio size from 1,000 to 10,000
for num_loans in list(range(1000,10000,1000)):

    reg_0 = test_investments(data_dict=data_dict,classifier=None,regressor=reg_ridge,strategy='Return-based',num_loans
                        output_to_file = True)
    result_sensitivity.append(reg_0['ret_PESS']['average return'])

result_sensitivity = np.array(result_sensitivity) * 100
sns.pointplot(np.array(list(range(1000,10000,1000))),result_sensitivity)
sns.despine()
plt.ylabel('Investment Return (%)',size = 14)
plt.xlabel('Portfolio Size',size = 14)
plt.show()
```



In [ ]:

In [58]:
```
## Test investment strategies using the best performing regressor

col_list = ['ret_PESS', 'ret_OPT', 'ret_INTa', 'ret_INTb']
test_strategy = 'Random'

print('strategy:',test_strategy)
# rf = RandomForestRegressor('max_depth'= 80, 'max_features'= 3, 'min_samples_leaf'= 3, 'min_samples_split'= 10, 'n_est
strat_rand = test_investments(data_dict=data_dict,classifier=None,regressor=None,strategy = test_strategy,num_loans = 1
                        output_to_file = False)

for ret_col in col_list:
    print(ret_col + ': ' + str(strat_rand[ret_col]['average return']))
```

```
strategy: Random
ret_PESS: 0.3995525507240218
ret_OPT: 1.4828544262089054
ret_INTa: 0.4997465688634208
ret_INTb: 1.4303406873756608
```

In [67]:
```
# test_strategy = 'Default-based'

test_strategy = 'Default-based'
rf = RandomForestRegressor(max_depth= 80, max_features= 3, min_samples_leaf= 3,min_samples_split= 10, n_estimators=300)
print('strategy:',test_strategy)
strat_def = test_investments(data_dict=data_dict,classifier=ada,regressor=reg_rf,strategy=test_strategy,num_loans = 100
                        output_to_file = True)

for ret_col in col_list:
    print(ret_col + ': ' + str(strat_def[ret_col]['average return']))
```

```
strategy: Default-based
ret_PESS: 0.4022310027265071
ret_OPT: 1.6593710779590614
ret_INTa: 0.4985030915100768
ret_INTb: 1.42195494190104
```

In [68]:
```
test_strategy = 'Return-based'

print('strategy:',test_strategy)
strat_ret = test_investments(data_dict=data_dict,classifier=ada,regressor=reg_rf,strategy=test_strategy,num_loans = 100
                        output_to_file = True)
```

```
for ret_col in col_list:
    print(ret_col + ': ' + str(strat_ret[ret_col]['average return']))
```

```
strategy: Return-based
ret_PESS: 0.5046219395614759
ret_OPT: 1.5820429910493217
ret_INTa: 0.49846228525843356
ret_INTb: 1.4210117300193956
```

In [ ]:

In [34]:
```
test_strategy = 'Default-return-based'

## For the Default-return-based strategy we need to fit a new regressor with separate = True
rf = RandomForestRegressor()
cv_parameters = {'max_depth': [110],'n_estimators': [200]}
reg_separate = fit_regression(rf ,data_dict,
                           cv_parameters = cv_parameters ,
                           model_name = 'Random Forest Regression',separate=True,output_to_file=False)

print('strategy:',test_strategy)
strat_defret = test_investments(data_dict=data_dict,classifier=ada,regressor=reg_separate,strategy=test_strategy,num_lo
                        output_to_file = True)

for ret_col in col_list:
    print(ret_col + ': ' + str(strat_defret[ret_col]['average return']))
```

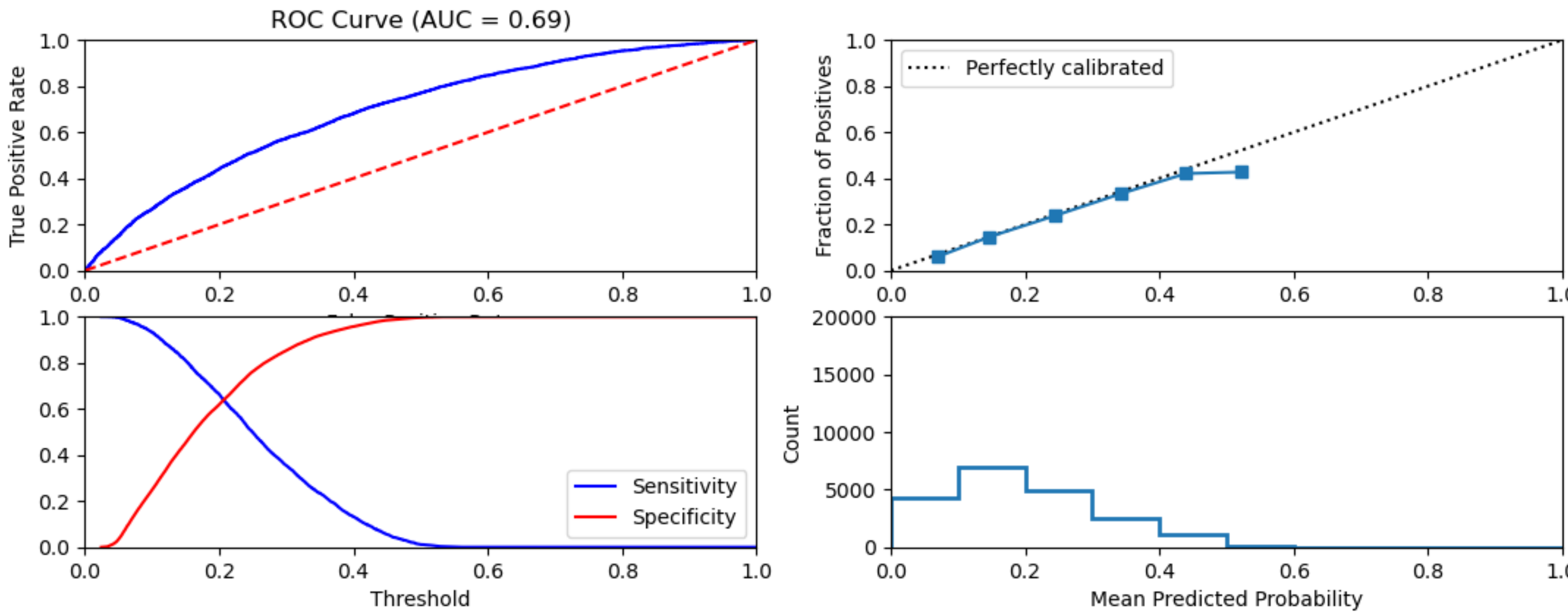
```
=========================================================
  Model: Random Forest Regression  Return column: ret_PESS
=========================================================
Fit time: 949.81 seconds
Optimal parameters:
model_0: {'max_depth': 110, 'n_estimators': 200} model_1 {'max_depth': 110, 'n_estimators': 200}

Testing r2 scores:
model_0: 0.061730031159804466
model_1: 0.11144342366523707
=========================================================
  Model: Random Forest Regression  Return column: ret_OPT
=========================================================
Fit time: 1040.62 seconds
Optimal parameters:
model_0: {'max_depth': 110, 'n_estimators': 200} model_1 {'max_depth': 110, 'n_estimators': 200}

Testing r2 scores:
model_0: -0.006178650829486543
model_1: -0.04628085286552075
=========================================================
  Model: Random Forest Regression  Return column: ret_INTa
=========================================================
Fit time: 293.46 seconds
Optimal parameters:
model_0: {'max_depth': 110, 'n_estimators': 200} model_1 {'max_depth': 110, 'n_estimators': 200}

Testing r2 scores:
model_0: -0.01015269323076673
model_1: 0.027609782972538133
=========================================================
  Model: Random Forest Regression  Return column: ret_INTb
=========================================================
Fit time: 246.28 seconds
Optimal parameters:
model_0: {'max_depth': 110, 'n_estimators': 200} model_1 {'max_depth': 110, 'n_estimators': 200}

Testing r2 scores:
model_0: 0.00034494312861921284
model_1: 0.03577256550639385
strategy: Default-return-based
ret_PESS: 0.3618433728097667
ret_OPT: 1.261324226737058
ret_INTa: 0.42241298909102565
ret_INTb: 1.2690405557025108
```

Question 6 - Train and Test YOURMODEL on the original data

In [86]:

```
data_dict = prepare_data()
cv_parameters = {'C': [0.1],
                 'max_iter': [100]}
l1_logistic = LogisticRegression(penalty='l1',solver = 'liblinear')
l1 = fit_classification(l1_logistic,
                        data_dict,
                        cv_parameters =cv_parameters ,
                        model_name ="L1 logistic classifier" )
```

```
==========================================================
  Model: L1 logistic classifier
==========================================================
Fit time: 2.52 seconds
Optimal parameters:
{'C': 0.1, 'max_iter': 100}

Accuracy-maximizing threshold was: 0.5042322964117306
Accuracy:  0.8087
              precision    recall  f1-score   support

  No default     0.8104    0.9968    0.8940     16183
     Default     0.4516    0.0110    0.0215      3817

    accuracy                         0.8087     20000
   macro avg     0.6310    0.5039    0.4577     20000
weighted avg     0.7419    0.8087    0.7275     20000
```




```
Similarity to LC grade ranking:  0.7921189332668903
Brier score: 0.14312416578569165
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.80266667]
```