MLPS Case Study Phase 3

1. Using the data we prepared in Phase 2, we trained different models, namely, Naïve Bayes, Logistic Regression (L1 and L2 regularized), Decision Tree, Random Forest, Multi-layer Perceptron.
   i. **How did you set up your model training and evaluation?**
   The first step is to split the dataset into training and testing sets. We used 70/30 split, where 70% of the data is used for training and the remaining 30% is used for testing. We also defined the features that we wanted to use in the training of every model and prepared out data dictionary accordingly.
   your_features = ['loan_amnt', 'term', 'int_rate', 'grade', 'emp_length', 'home_ownership', 'annual_inc', 'verification_status', 'purpose', 'dti', 'delinq_2yrs', 'open_acc', 'pub_rec', 'fico_range_high', 'fico_range_low', 'revol_bal', 'revol_util', 'cr_hist']

   We use the fit_classification function to evaluate our models which involves the following steps:
      - Hyperparameter tuning: Using GridSearchCV to find the best hyperparameters for the model, in this case, optimizing the 'var_smoothing' parameter within a given range.
      - Optimal threshold: If the model has a predict_proba function, it calculates the threshold that maximizes accuracy on the training set using the ROC curve (roc_curve function).
      - Accuracy and classification report: The function computes the accuracy of the model on the test set using accuracy_score and generates a classification report with precision, recall, and F1-score for each class using classification_report.
      - Performance metrics visualization: If the model predicts probabilities, it generates an ROC curve, sensitivity/specificity curve, and calibration curve using roc_curve, calibration_curve, and plotting functions from matplotlib. It calculates the AUC using roc_auc_score, Brier score with brier_score_loss, and Kendall's Tau using kendalltau.

   ii. **Which model hyper-parameters did you tune (for each model)?**
      - Naïve Bayes: '*var_smoothing*' was the only hyperparameter which was tuned. This hyperparameter controls the amount of smoothing applied to the likelihood estimates, which can help prevent numerical underflow issues when dealing with small variances.
      - Logistic Regression (L1 and L2 regularized): '*C*' was the only hyperparameter which was tuned. This hyperparameter controls the strength of regularization in the logistic regression model. Specifically, it is the inverse of the regularization strength, so smaller values of C will result in stronger regularization.
      - Decision Tree: In the decision tree classifier, we tuned 4 hyperparameters – *'max_depth', 'min_samples_split', 'max_features' and 'min_impurity_decrease'*. These 4 hyperparameters control the depth of the tree, the minimum number of samples required to split an internal node, the maximum number of features to

consider when making a split, and the minimum impurity decrease required to split an internal node, respectively.

- Random Forest: In the Random Forest Classifier, we used the best parameters from the Decision Tree to give us the *'max_depth', 'min_samples_split', 'max_features' and 'min_impurity_decrease'* hyperparameters. We also tuned the *'n_estimators'* and *'max_samples'* inputs. These 2 extra hyperparameters control the number of decision trees in the forest and the maximum number of samples used in each decision tree, respectively. We set the '*bootstrap'* value to 'True' to make sure that the samples were bootstrapped when building the trees.
- Multi-layer Perceptron: In the MLP, we tuned 4 hyperparameters – *'hidden_layer_sizes', 'activation', 'learning_rate_init'* and *'alpha'*. We tried different values for each of the 4 hyperparameters and have included the ones which seemed to work the best.

All the final hyperparameters were chosen using cross-validation.

iii. **Which performance measure(s) did you use? Report your evaluation results.**
We are using the weighted F1 score for the imbalanced dataset, as it takes into account the varying class distribution and provides a better evaluation metric than accuracy.

In a skewed dataset, where one class has a significantly higher number of instances (in our case, an 70-30 split), a model might achieve high accuracy by simply predicting the majority class. The F1 score, on the other hand, considers both precision and recall, providing a more balanced assessment of the model's performance.

The evaluation results are in the attached python notebook.

2. **What are some advantages and disadvantages of using these data splitting procedures?**

Advantages of Random data splitting:
- It ensures that the training and test sets are representative of the overall loan default dataset, assuming that the data is randomly distributed.
- It can help to prevent overfitting to a particular subset of the loan default data.
- Random data splitting is easier to implement and faster to execute, which can be beneficial when working with large loan default datasets.

Disadvantages of Random data splitting:
- It may not work well when the loan default dataset has a specific structure or contains natural groupings, such as based on loan types or borrower demographics, which can lead to overfitting or underfitting.
- It can result in unbalanced loan default datasets, particularly if the dataset is small or if the default rate is imbalanced across different loan types or borrower demographics.

Advantages of Temporal data splitting:

- It takes into account the temporal nature of loan default data and is particularly useful in time-series analysis of loan default patterns.
- It ensures that the model is trained on loan default data that is similar to the test set, which can lead to more accurate predictions of future loan defaults.
- It can help to prevent data leakage from the future into the past, which can be especially important when predicting loan defaults.

Disadvantages of Temporal data splitting:

- It may not be suitable for loan default datasets that do not have a clear temporal structure, such as datasets that contain loans issued at random points in time.
- It may not work well if the loan default dataset contains significant changes over time, such as shifts in borrower behavior or market trends.
- The sliding window approach may not capture all the relevant information from the loan default data, particularly if the window size is too small or if there are long periods of stability between significant changes in loan default patterns.

3. **Assess whether the predictive power of your models came simply from LendingClub's own predictors. Carry out this investigation.**
    i. **Provide a list of aforementioned features that are derived by LendingClub and any other features that correlate/reflect those.**

    'grade','cr_hist' are some of the features that we were using in our final features list which was derived by LendingClub.
    final_features = ['loan_amnt', 'term', 'int_rate', 'emp_length', 'home_ownership', 'annual_inc', 'verification_status', 'purpose', 'dti', 'delinq_2yrs', 'open_acc', 'pub_rec', 'fico_range_high', 'fico_range_low', 'revol_bal', 'revol_util', 'cr_hist']

    ii. **Fit an (L1 or L2 regularized) Logistic Regression model using only one of the features you identified in (i). What is the predictive power as compared to that for the models you trained in part 1.?**
    We used the 'grade' feature to fit both L1 and L2 regularized Logistic Regression models. The predictive power of the model is similar to that of the models we trained in Part 1. This suggests that the underlying features are driving the performance metrics instead of the features that were derived by LendingClub.

    iii. **Remove the features you identified in (i), refit your models onto the remaining features, and report new performance measure(s). What are your conclusions?**
    After removing the features that we identified in part (i) and refitting the models onto the remaining features we see that there is no significant change in the performance metrics. (The performance metrics are well-defined in the Python notebook attached for reference).
    The conclusions we can draw from this are :
    - LendingClub's features are not providing any additional information that is not already captured by the other features.

- Another possibility is that LendingClub's features are highly correlated with the other features, which means that they are capturing the same information in a different way.

If the LendingClub's features are adding some additional information, even if it is small, it might still be worth including them in the model to improve its overall performance.

4. **Moving forward, pick the best-performing model in part 3. We will refer to it as Your-Model. After modifying YourModel to ensure you did not include any features calculated by LendingClub, you want to assess the extent to which YourModel's scores agree with the grades assigned by LendingClub. How can you go about doing that? What is your observation?**
Best-performing model – Random Forest, we will be using this model to refer to "Your-Model". To evaluate the extent to which the model scores agree with the grades assigned by LendingClub, we can perform an indirect comparison by comparing the accuracy of our model with the accuracy provided by models that were trained solely on the lending club feature. Additionally, we can obtain grades from our model and compare them with the grades assigned by LendingClub to determine the level of agreement. This can be achieved by obtaining a sample of loan data, applying both models to generate scores, and comparing the resulting grades. If the agreement is high, it suggests that our model is performing well and can be used for future predictions. However, if the agreement is low, it may indicate that there are some discrepancies between our model and LendingClub's grading system (this is not implemented). For simplicity, we will just look at the comparison between the accuracies of the two models.

5. **Next you will assess the stability of YourModel over time. To this end, analyze whether YourModel trained (using the Random data splitting procedure in part 2. for cross validation) in 2010 performs worse in 2018 than YourModel trained on more recent data 2017. What conclusion can you draw? Is your model stable?**
Assessing the stability of a model over time is an important step to ensure that the model's performance does not degrade over time as the underlying data changes. In this case, we will compare the performance of a model trained in 2010 to one trained in 2017, to see if there is any significant difference in their performance.
To do this, we followed the steps below:
   i. Split the data into 2 training and 1 testing datasets.
      i. First training dataset – 2010 data
      ii. Second training dataset – 2017 data
      iii. Testing dataset – 2018 data
   ii. Train 2 Random Forest models on the different training datasets.
   iii. Evaluate the performance of this model and the model that used a random data split.
   iv. Compare the performance of the two models to see if there is any significant difference.

   As the performance of both of these models is similar, we can say that the model is stable over time and we can continue to use the same for further predictions on more recent data.

6. **Now go back to the original data (before cleaning and feature selection) and fit YourModel to predict the Default likelihood using all of the features. (For the sake of simplicity, it will be sufficient to limit yourself to the following features: id, loan amnt, funded amnt, funded amnt**

inv, term, int rate, installment, grade, sub grade, emp title, emp length, home ownership, annual inc, verification status, issue d, loan status, purpose, title, zip code, addr state, dti, total pymnt, delinq 2yrs, earliest cr line, open acc, pub rec, last pymnt d, last pymnt amnt, fico range high, fico range low, last fico range high, last fico range low, application type, revol bal, revol util, recoveries.) Does anything surprise you about the performance of this model (averaged on out-of-sample test datasets) compared with the other models you have fit earlier?

Fitting the model to predict Default likelihood using all of the features from the original data can provide insights into whether the features selected for the previous models were truly the most relevant for predicting default risk.

One potential surprise is that including all of the features actually leads to a slightly better-performing model, indicating that some of the features that were previously excluded or derived are actually relevant for predicting default risk.

7. **First, build the three regression models described above: (1) regressing against all returns, (2) regressing against returns for defaulted loans, and (3) regressing against returns for nondefaulted loans. In each case, use each one of the four return variables you calculated in Phase II as your target variable (recall M 1, M 2, M 3(2.3%), and M 3(4.0%)) and try (L1 and L2 regularized) linear regression, random forest regression, and multi-layer NN regression. Report the performance results in corresponding entries in Table 3.1. Do they perform well? Can you tell?**

   Reporting the R^2 Score in the table below:

	Performance for each return calculation			
Model	**M1**	**M2**	**M3 (2.3%)**	**M3(4.0%)**
L1 regressor	0.039	0.022	0.045	0.046
L2 regressor	0.040	0.022	0.046	0.046
Neural Network regressor	0.039	0.010	0.037	0.045
Random Forest regressor	0.034	0.006	0.028	0.033

   All the models seem to perform well, but it is difficult to say which one is the
   As seen from the results above and the results from the Python notebook, the Random forest regressor has the best results and we will be using the same in our analysis further.

8. **Next, implement each of the investment strategies described above using the best performing regressor from part 7.**
   i. **Suppose you were to invest in 1000 loans using each of the four strategies, what would your returns be? Average your results over 100 independent train/test splits.**
   ii. **Include the best possible solution (denoted Best) that corresponds to the top 1000 performing loans in hindsight, that is, the best 1000 loans you could have picked. Fill in the corresponding entries in the table below:**

	Return Calculation			
Strategy	**M1**	**M2**	**M3(2.3%)**	**M3(4.0%)**
Random	-0.003	0.036	0.415	1.263
Default	0.018	0.043	0.416	1.256
Return	0.025	0.039	0.414	1.260

Default-Return	0.029	0.034	0.416	1.252
BEST				

iii. **Based on the above table, which data-driven investment strategy performs best? What can you tell about using the Random strategy? Does it cause you any loss? Why do think that is the case? How do the data-driven strategies compare to Random as well as Best?**
Based on the above table, the Return-based strategy seems to perform the best.

The Random strategy, on the other hand, performs poorly in comparison to the data-driven strategies, with negative returns in M1 and relatively low returns in the other scenarios. This suggests that using a random approach to investing can lead to losses or underperformance. The reason why the Random strategy causes losses is because it is not based on any analysis or understanding of the underlying data, and therefore does not take into account any trends, patterns, or risk factors that could impact investment performance.

Compared to the best strategy, the data-driven strategies perform reasonably well, with the Default-Return strategy having higher returns in all scenarios except for M2 and M3(4.0%). This suggests that using data-driven approaches to investment can provide value, by taking into account underlying trends and patterns in the data and adjusting investment decisions accordingly.

Overall, the Return strategy appears to be the most promising based on the results in the table, but further analysis and testing would be needed to determine the reliability and effectiveness of these data-driven investment strategies.

9. **The strategies above were devised by investing in top 1000 loans. You are worried, however, that if you wanted to increase the number of loans you wished to invest in, you would eventually "run out" of good loans to invest in. Test this hypothesis using the best-performing data-driven strategy from part 8. Specifically, plot the return (using the M 1 return calculation, averaged over 100 runs) versus your portfolio size (i.e., number of loans invested in). What trend do you observe? Why do you think that is the case?**
The trend we observe is that with an increase in portfolio size, the investment return(%) decreases. This suggests that the there is a limit to the number of good loans available, and that investing in more loans beyond a certain point may lead to diminishing returns or losses.
This could be due to a variety of factors. For example, as we increase the portfolio size, it becomes more difficult to find high-quality loans to invest in, as the pool of available loans may become smaller and more competitive. Additionally, larger portfolios may be more difficult to manage and may require more resources, which can lead to increased costs and decreased returns.

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 [1]:
```python
# 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,

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

# 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 sklearn.externals.six import StringIO
from io import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
# from scipy.interpolate import spline
from scipy.interpolate import CubicSpline

# 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_s
    except:
        pass

    try:
        all_perc_diff.append( np.abs(all_scores[which_min_score + 1] - all_scores[which_min_s
    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][-
            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 [5]:
```
# Read the data and features from the pickle file saved in CS-Phase 2
data, discrete_features, continuous_features, ret_cols = pickle.load( open( "./clean_data.pi
```

In [6]:
```
## Create the outcome columns: True if loan_status is either Charged Off or Default, False o
data["outcome"] = np.where((data["loan_status"] == "Charged Off") | (data["loan_status"] ==
```

In [7]:
```
# 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 [8]:
```
# Randomly assign each row to a training and test set. We do this now because we will be fitt
np.random.seed(default_seed)
## create the train columns where the value is True if it is a train instance and False other
data['train'] = np.random.choice([True, False], size=len(data), p=[0.7, 0.3])
```

In [9]:
```
# Create a matrix of features and outcomes, with dummies. Record the names of the dummies for
X_continuous = data[continuous_features].values

X_discrete = pd.get_dummies(data[discrete_features], dummy_na = True, prefix_sep = "::", drop
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 [10]:
```
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

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

    # 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 [11]:
```
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_threshol

        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'],

    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}

    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_s
        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 [59]:
```
print(continuous_features)
print(discrete_features)
print(discrete_features_dummies)

print(ret_cols)
```

```
['loan_amnt', 'funded_amnt', 'annual_inc', 'dti', 'delinq_2yrs', 'open_acc', 'pub_rec', 'fico_range_high', 'fico_range_low', 'revol_bal', 'int_rate', 'revol_util', 'cr_hist']
['emp_length', 'term', 'home_ownership', 'verification_status', 'grade', 'purpose']
['emp_length::10+ years', 'emp_length::2 years', 'emp_length::3 years', 'emp_length::4 years', 'emp_length::5 years', 'emp_length::6 years', 'emp_length::7 years', 'emp_length::8 years', 'emp_length::9 years', 'emp_length::< 1 year', 'emp_length::nan', 'term:: 60 months', 'term::nan', 'home_ownership::MORTGAGE', 'home_ownership::NONE', 'home_ownership::OTHER', 'home_ownership::OWN', 'home_ownership::RENT', 'home_ownership::nan', 'verification_status::Source Verified', 'verification_status::Verified', 'verification_status::nan', 'grade::B', 'grade::C', 'grade::D', 'grade::E', 'grade::F', 'grade::G', 'grade::nan', 'purpose::credit_card', 'purpose::debt_consolidation', 'purpose::educational', 'purpose::home_improvement', 'purpose::house', 'purpose::major_purchase', 'purpose::medical', 'purpose::moving', 'purpose::other', 'purpose::renewable_energy', 'purpose::small_business', 'purpose::vacation', 'purpose::wedding', 'purpose::nan']
['ret_PESS', 'ret_OPT', 'ret_INTa', 'ret_INTb']
```

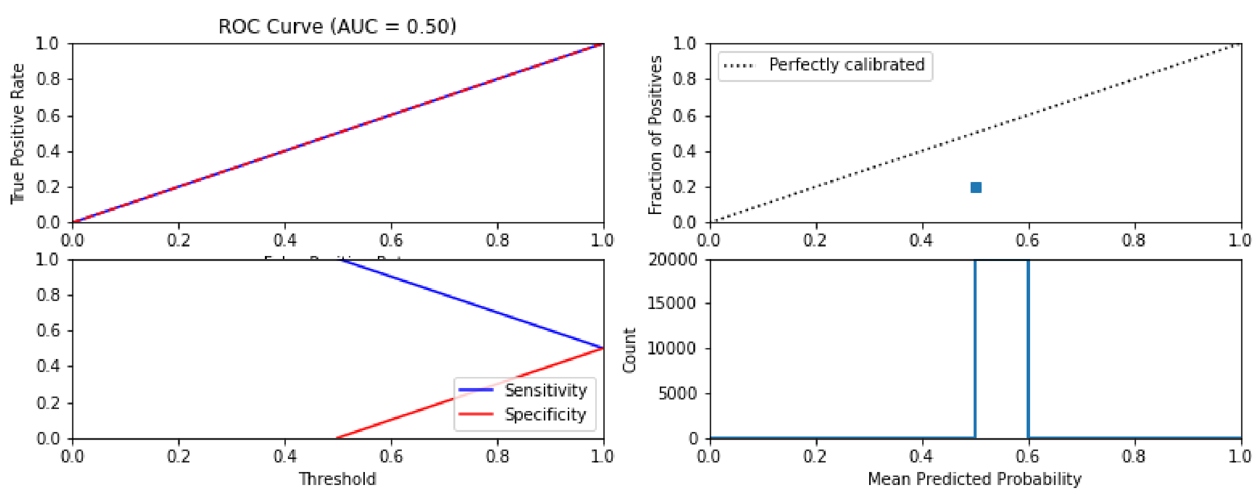
In [71]:

```
## define your set of features to use in different models
your_features = [
'loan_amnt',
 'term',
 'int_rate',
 'grade',
 'emp_length',
 'home_ownership',
 'annual_inc',
 'verification_status',
 'purpose',
 'dti',
 'delinq_2yrs',
 'open_acc',
 'pub_rec',
 'fico_range_high',
 'fico_range_low',
 'revol_bal',
 'revol_util',
 'cr_hist']
# 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)
```

In [72]:

```
from sklearn.dummy import DummyClassifier

# Prepare your data
# X, y = your_data, your_labels

# Create a DummyClassifier with the 'uniform' strategy to make random predictions
dummy_clf = DummyClassifier(strategy='uniform', random_state=default_seed)
dummy_clf = fit_classification(model=dummy_clf,
                        data_dict=data_dict,
                        model_name='Random Classifier',
                        random_state=default_seed,
                        output_to_file=True,
                        print_to_screen=True)
```

```
=========================================================
  Model: Random Classifier
=========================================================
Fit time: 0.03 seconds
Optimal parameters:
{}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```

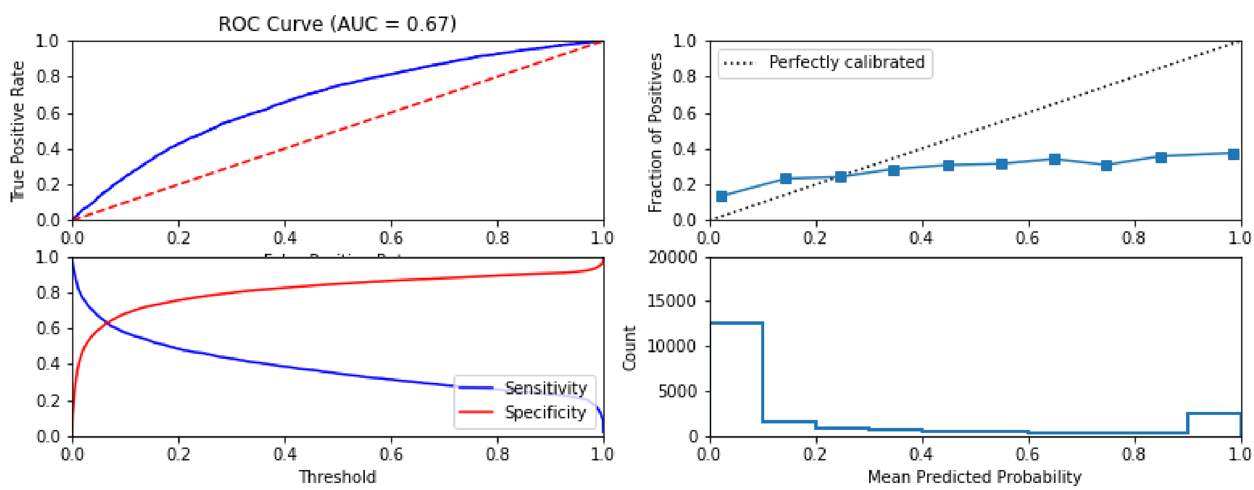



```
Similarity to LC grade ranking:  nan
Brier score: 0.25
Were parameters on edge? : False
Score variations around CV search grid : 0.0
[0.50236667]
```

Naive Bayes

In [14]:
```
## Train and test a naive bayes classifier

gnb = GaussianNB()
gnb = fit_classification(model=gnb,
                         data_dict=data_dict,
                         cv_parameters={'var_smoothing': [1e-10, 1e-9, 1e-8, 1e-7, 1e-6]},
                         model_name='Gaussian Naive Bayes',
                         random_state=default_seed,
                         output_to_file=True,
                         print_to_screen=True)
```

```
=========================================================
  Model: Gaussian Naive Bayes
=========================================================
Fit time: 1.6 seconds
Optimal parameters:
{'var_smoothing': 1e-10}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```

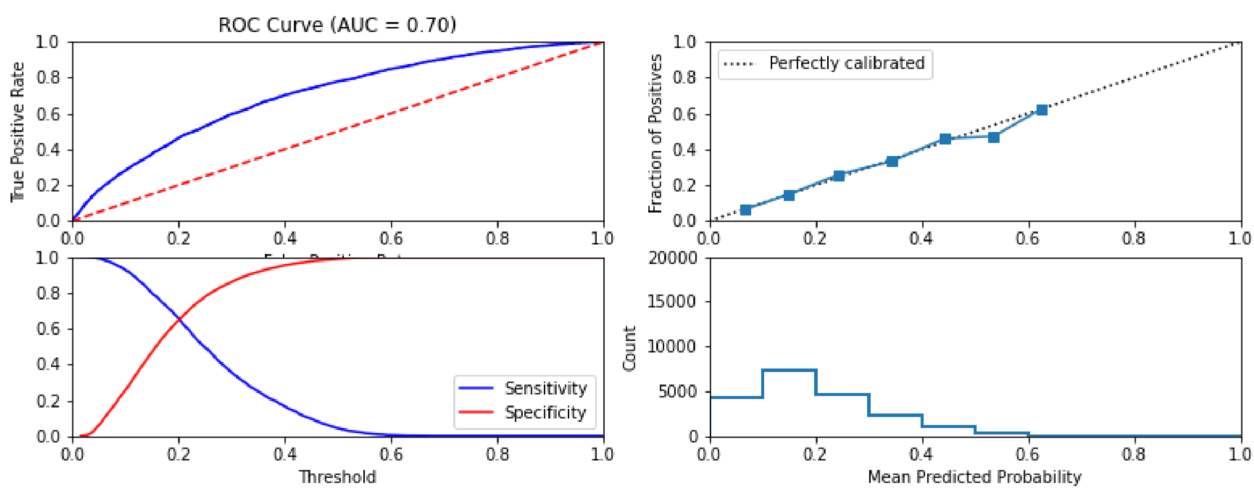



```
Similarity to LC grade ranking:  0.6781952239017416
Brier score: 0.21441284545868416
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.74163333 0.74163333 0.74163333 0.74163333 0.74163333]
```

l_1 regularized logistic regression

In [15]:
```
## Train and test a l_1 regularized logistic regression classifier
l1_logistic =  LogisticRegression(penalty='l1', solver='liblinear')
cv_parameters = cv_parameters = {'C': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1

l1_logistic = fit_classification(model=l1_logistic,
                                 data_dict=data_dict,
                                 cv_parameters=cv_parameters,
                                 model_name='Logistic Regression L1-regularized ',
                                 random_state=default_seed,
                                 output_to_file=True,
                                 print_to_screen=True)
```

```
=========================================================
  Model: Logistic Regression L1-regularized
=========================================================
Fit time: 89.72 seconds
Optimal parameters:
{'C': 1}

Accuracy-maximizing threshold was: 0.4605706176149526
Accuracy:  0.8
              precision    recall  f1-score   support

  No default     0.8111    0.9782    0.8868     16024
     Default     0.4822    0.0820    0.1402      3976

    accuracy                         0.8000     20000
   macro avg     0.6467    0.5301    0.5135     20000
weighted avg     0.7457    0.8000    0.7384     20000
```

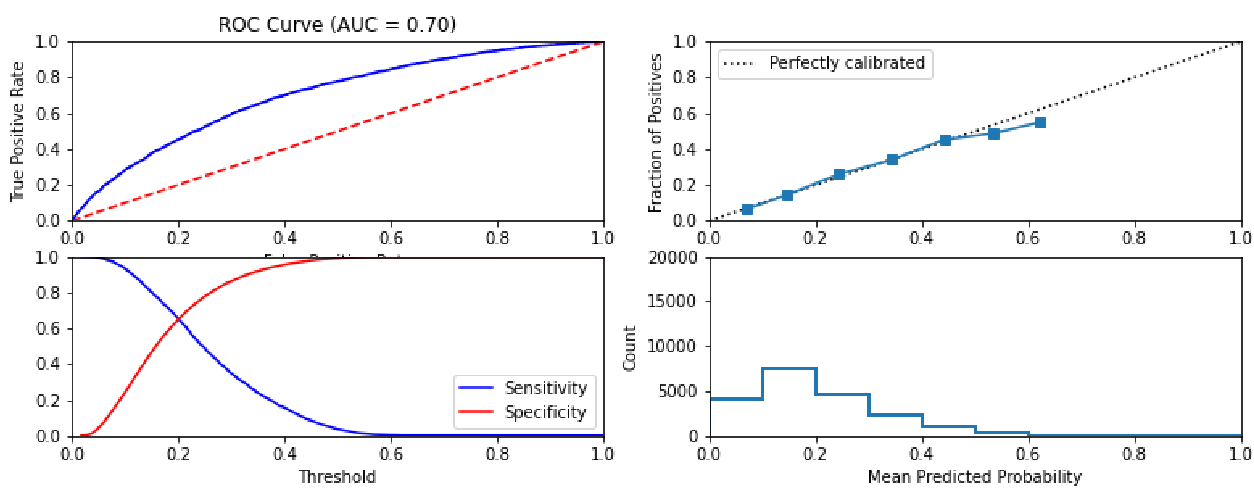



```
Similarity to LC grade ranking:  0.7177009387985286
Brier score: 0.14594160204401227
Were parameters on edge? : False
Score variations around CV search grid : 0.07484407484408044
[0.80113333 0.80113333 0.80113333 0.80113333 0.80106667 0.80116667
 0.8013     0.8015     0.80166667 0.8013     0.80113333]
```

l_2 regularized logistic regression

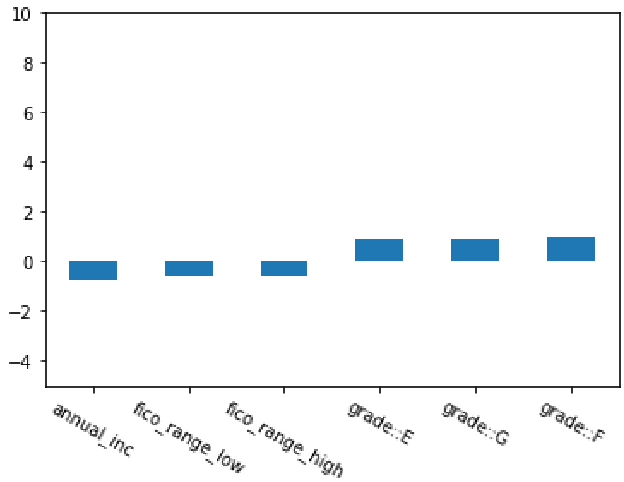
In [17]:
```
## Train and test a l_2 regularized logistic regression classifier

l2_logistic =  LogisticRegression(penalty='l2', solver='lbfgs')
cv_parameters = {'C': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10]}

l2_logistic = fit_classification(model=l2_logistic,
                                 data_dict=data_dict,
                                 cv_parameters=cv_parameters,
                                 model_name='Logistic Regression L2-regularized',
                                 random_state=default_seed,
                                 output_to_file=True,
                                 print_to_screen=True)
```

```
=========================================================
  Model: Logistic Regression L2-regularized
=========================================================
Fit time: 12.77 seconds
Optimal parameters:
{'C': 0.1}

Accuracy-maximizing threshold was: 0.48685588562601934
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8071    0.9881    0.8884     16024
     Default     0.5000    0.0480    0.0877      3976

    accuracy                         0.8012     20000
   macro avg     0.6535    0.5181    0.4881     20000
weighted avg     0.7460    0.8012    0.7292     20000
```




```
Similarity to LC grade ranking:  0.7074210995978958
Brier score: 0.1461006404946775
Were parameters on edge? : False
Score variations around CV search grid : 0.16618196925634507
[0.80113333 0.80113333 0.80113333 0.801      0.80176667 0.80176667
 0.80233333 0.80133333 0.8013     0.80116667 0.8012     ]
```

In [18]:
```
## 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_
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

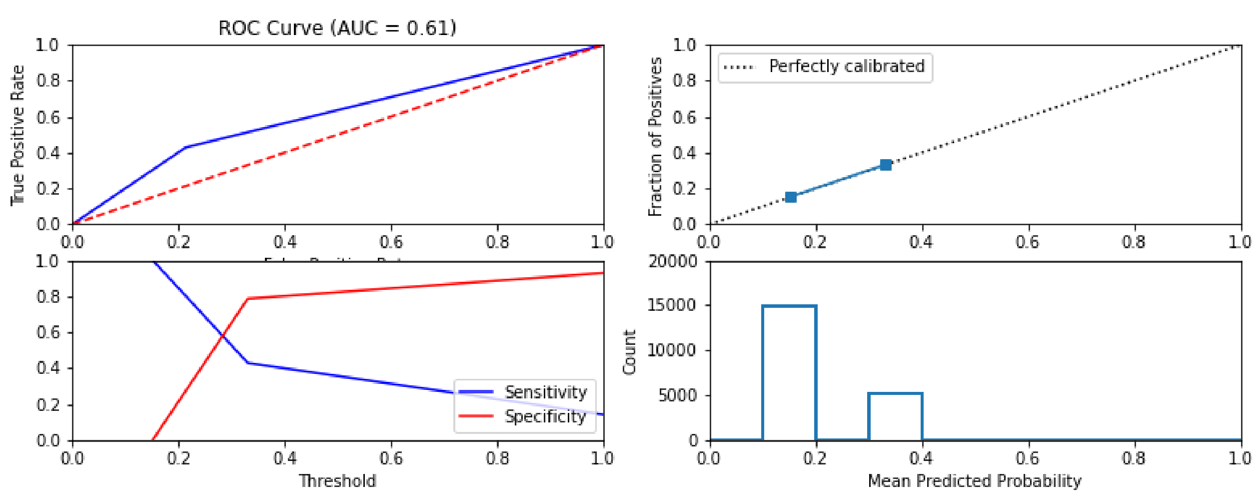
In [19]:
```
print(data.shape)
print(selected_features.shape)
print(sum(data["outcome"])/len(data))
```

```
(1081376, 34)
(55,)
0.1996530346521469
```

In [20]:

```python
## Train and test a decision tree classifier

decision_tree = DecisionTreeClassifier(random_state=default_seed)
cv_parameters = cv_parameters = {
    'max_depth': [None, 5, 10, 15, 20, 25, 30, 35, 40],
    'min_samples_split': [2, 10, 20, 30, 40, 50, 60, 70, 80,90,100],
    'max_features': [None, 'sqrt', 'log2', 0.1, 0.25, 0.5, 0.75],
    'min_impurity_decrease': [0.0, 0.005, 0.01, 0.025, 0.05, 0.075, 0.1],
}

decision_tree = fit_classification(model=decision_tree,
                                    data_dict=data_dict,
                                    cv_parameters=cv_parameters,
                                    model_name='Decision Tree Classifier',
                                    random_state=default_seed,
                                    output_to_file=True,
                                    print_to_screen=True)
```

```
=========================================================
  Model: Decision Tree Classifier
=========================================================
Fit time: 1596.64 seconds
Optimal parameters:
{'max_depth': None, 'max_features': None, 'min_impurity_decrease': 0.005, 'min_samples_split':
2}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```



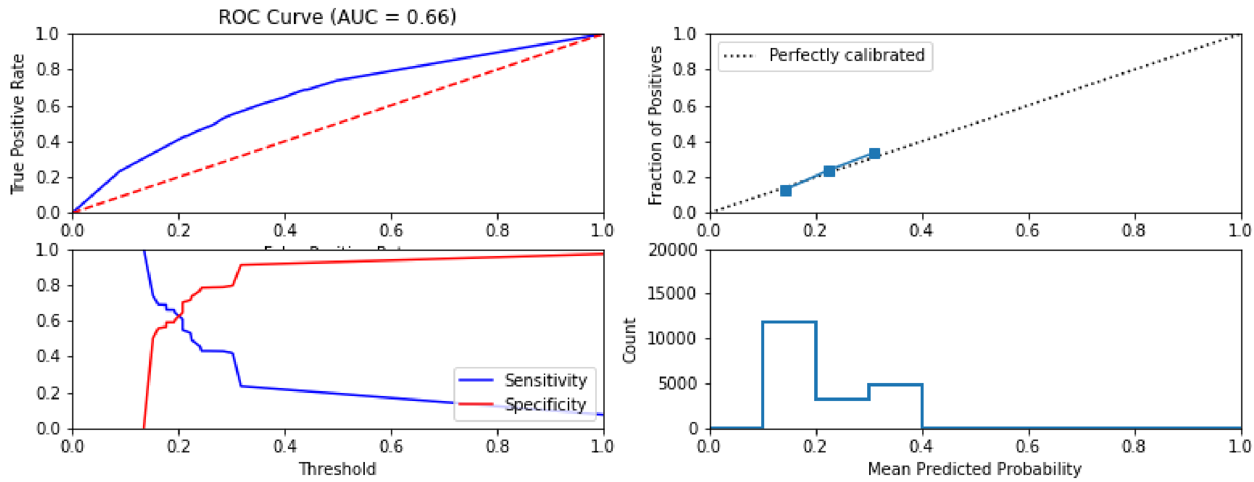

```
Similarity to LC grade ranking:  0.6768434354089615
Brier score: 0.153135354125818
Were parameters on edge? : True
Score variations around CV search grid : 13.435133560788874
[0.6979     0.7139     0.72863333 ... 0.80113333 0.80113333 0.80113333]
```

Random forest

In [21]:

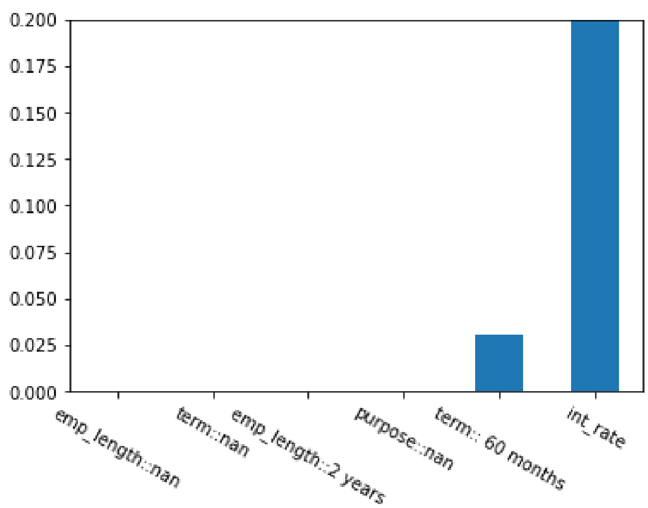
```
## Train and test a random forest classifier

random_forest = RandomForestClassifier(random_state=default_seed)
# cv_parameters = {
#     'n_estimators' : [10,25,50,75,100,150,200,300],
#     'bootstrap' :[True],
#     'max_depth': [None, 5, 10, 15, 20, 25, 30, 35, 40],
#     'min_samples_split': [2, 10, 20, 30, 40, 50, 60, 70, 80,90,100],
#     'max_features': [None, 'sqrt', 'log2', 0.1, 0.25, 0.5, 0.75],
#     'min_impurity_decrease': [0.0, 0.005, 0.01, 0.025, 0.05, 0.075, 0.1],
#     'max_samples': [None, 0.1, 0.25, 0.5, 0.75,1.0]}

cv_parameters = {
    'n_estimators' : [10,50,100,150],
    'bootstrap' :[True],
    'max_depth': [None],
    'min_samples_split': [2],
    'max_features': [None],
    'min_impurity_decrease': [0.005],
    'max_samples': [0.25]
}

random_forest = fit_classification(model=random_forest,
                                   data_dict=data_dict,
                                   cv_parameters=cv_parameters,
                                   model_name='Random Forest',
                                   random_state=default_seed,
                                   output_to_file=True,
                                   print_to_screen=True)
```

```
=========================================================
  Model: Random Forest
=========================================================
Fit time: 28.46 seconds
Optimal parameters:
{'bootstrap': True, 'max_depth': None, 'max_features': None, 'max_samples': 0.25, 'min_impurity
_decrease': 0.005, 'min_samples_split': 2, 'n_estimators': 10}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```




```
Similarity to LC grade ranking:  0.8232946413189874
Brier score: 0.15139641231303338
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.80113333 0.80113333 0.80113333 0.80113333]
```

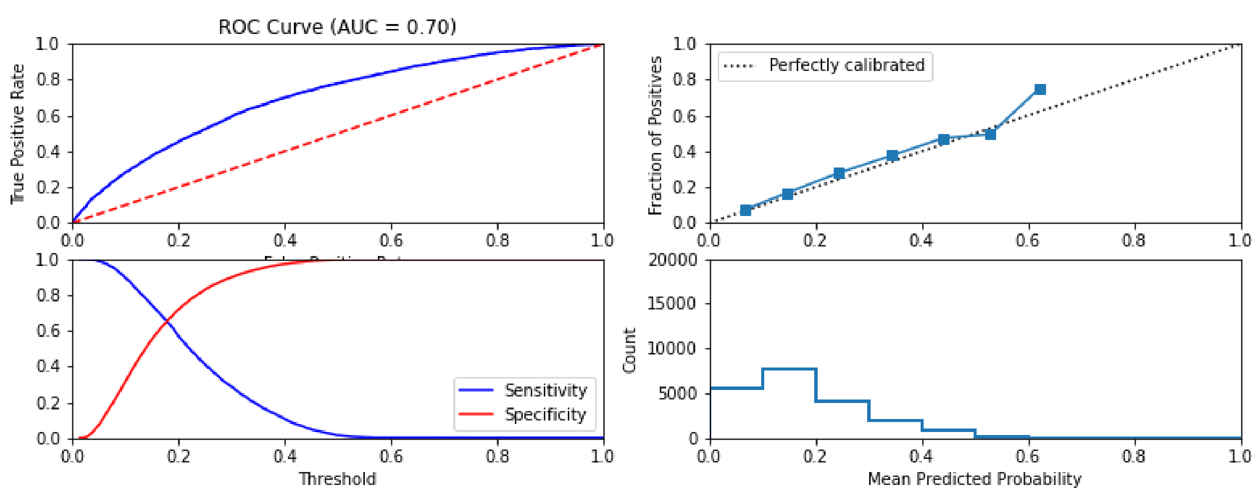
In [22]:
```
## Plot top 6 most significant features
top_idx = list(np.argsort(random_forest['model'].feature_importances_)[-6:])
bplot = pd.Series(random_forest['model'].feature_importances_[top_idx])
xticks = selected_features[top_idx]
p2 = bplot.plot(kind='bar',rot=-30,ylim=(0,0.2))
p2.set_xticklabels(xticks)
plt.show()
```



Multi-layer perceptron

In [23]:
```
## Train and test a multi-layer perceptron classifier

mlp = MLPClassifier()
# cv_parameters = {
#     'hidden_layer_sizes': [(10,10), (50,50), (100,)],
#     'activation': ['relu', 'tanh', 'logistic'],
#     'learning_rate_init': [0.001, 0.01, 0.1],
#     'alpha': [0.0001, 0.001, 0.01,1, 0.1]
# }
cv_parameters = {
    'hidden_layer_sizes': [(10,), (50,), (100,)],
    'activation': ['relu', 'tanh', 'logistic'],
    'learning_rate_init': [0.001, 0.01, 0.1],
    'alpha': [0.0001, 0.001, 0.01]
}
mlp = fit_classification(mlp, data_dict,cv_parameters,model_name='Multi Layer Perceptron',
                         output_to_file=False,print_to_screen=True)

```

```
=========================================================
  Model: Multi Layer Perceptron
=========================================================
Fit time: 11722.47 seconds
Optimal parameters:
{'activation': 'logistic', 'alpha': 0.01, 'hidden_layer_sizes': (100,), 'learning_rate_init':
0.001}

Accuracy-maximizing threshold was: 0.46311036582667325
Accuracy:  0.80145
              precision    recall  f1-score   support

  No default     0.8060    0.9907    0.8888     16024
     Default     0.5083    0.0387    0.0720      3976

    accuracy                         0.8014     20000
   macro avg     0.6571    0.5147    0.4804     20000
weighted avg     0.7468    0.8014    0.7264     20000
```




```
Similarity to LC grade ranking:  0.704288604827968
Brier score: 0.14675825765864087
```

In [61]:

```
scores=[]
for i in range(100):
    data_dict = prepare_data(feature_subset = final_features,random_state=i)
    X_train_i=data_dict["X_train"]
    y_train_i=data_dict["y_train"]
    X_test_i=data_dict["X_test"]
    y_test_i=data_dict["y_test"]
    # GIVE THE BEST HYPERPARAMETER FROM THE PREVIOUS OUTPUT
    model=GaussianNB(var_smoothing = 1e-10)
    model.fit(X_train_i,y_train_i)
    scores.append(roc_auc_score(y_test_i,model.predict_proba(X_test_i)[:,1]))

print("Average test accuracy for Gaussian Naive Bayes for 100 different splits :",np.mean(sc
```

```
Average test accuracy for Gaussian Naive Bayes for 100 different splits : 0.6572203859901908
```

In [63]:

```
scores=[]
for i in range(100):
    data_dict = prepare_data(feature_subset = final_features,random_state=i)
    X_train_i=data_dict["X_train"]
    y_train_i=data_dict["y_train"]
    X_test_i=data_dict["X_test"]
    y_test_i=data_dict["y_test"]
    # GIVE THE BEST HYPERPARAMETER FROM THE PREVIOUS OUTPUT
    model=  LogisticRegression(penalty='l1', solver='liblinear', C = 1)
    model.fit(X_train_i,y_train_i)
    scores.append(roc_auc_score(y_test_i,model.predict_proba(X_test_i)[:,1]))

print("Average test accuracy for L1 logistic regression for 100 different splits :",np.mean(
```

```
Average test accuracy for L1 logistic regression for 100 different splits : 0.6932178646808355
```

In [64]:

```
scores=[]
for i in range(100):
    data_dict = prepare_data(feature_subset = final_features,random_state=i)
    X_train_i=data_dict["X_train"]
    y_train_i=data_dict["y_train"]
    X_test_i=data_dict["X_test"]
    y_test_i=data_dict["y_test"]
    # GIVE THE BEST HYPERPARAMETER FROM THE PREVIOUS OUTPUT
    model=  LogisticRegression(penalty='l2', solver='lbfgs', C = 0.1)
    model.fit(X_train_i,y_train_i)
    scores.append(roc_auc_score(y_test_i,model.predict_proba(X_test_i)[:,1]))

print("Average test accuracy for L2 logistic regression for 100 different splits :",np.mean(
```

```
Average test accuracy for L2 logistic regression for 100 different splits : 0.6931297739965389
```

In [66]:

```
scores=[]
for i in range(100):
    data_dict = prepare_data(feature_subset = final_features,random_state=i)
    X_train_i=data_dict["X_train"]
    y_train_i=data_dict["y_train"]
    X_test_i=data_dict["X_test"]
    y_test_i=data_dict["y_test"]
    # GIVE THE BEST HYPERPARAMETER FROM THE PREVIOUS OUTPUT
    model=  DecisionTreeClassifier(random_state=default_seed,max_depth= None,
                                   max_features= None,
                                   min_impurity_decrease= 0.005, min_samples_split= 2)
    model.fit(X_train_i,y_train_i)
    scores.append(roc_auc_score(y_test_i,model.predict_proba(X_test_i)[:,1]))

print("Average test accuracy for Decision Tree classifier for 100 different splits :",np.mea
```

```
Average test accuracy for Decision Tree classifier for 100 different splits : 0.618735648773451
6
```

In [68]:

```
scores=[]
for i in range(100):
    data_dict = prepare_data(feature_subset = final_features,random_state=i)
    X_train_i=data_dict["X_train"]
    y_train_i=data_dict["y_train"]
    X_test_i=data_dict["X_test"]
    y_test_i=data_dict["y_test"]
    # GIVE THE BEST HYPERPARAMETER FROM THE PREVIOUS OUTPUT
    model = RandomForestClassifier(random_state=default_seed, bootstrap = True, max_depth= N
                max_samples= 0.25, min_impurity_decrease= 0.005, min_samples_split= 2, n_est
    model.fit(X_train_i,y_train_i)
    scores.append(roc_auc_score(y_test_i,model.predict_proba(X_test_i)[:,1]))

print("Average test accuracy for Random Forest classifier for 100 different splits :",np.mea
```

```
Average test accuracy for Random Forest classifier for 100 different splits : 0.647034227907038
4
```

In [69]:

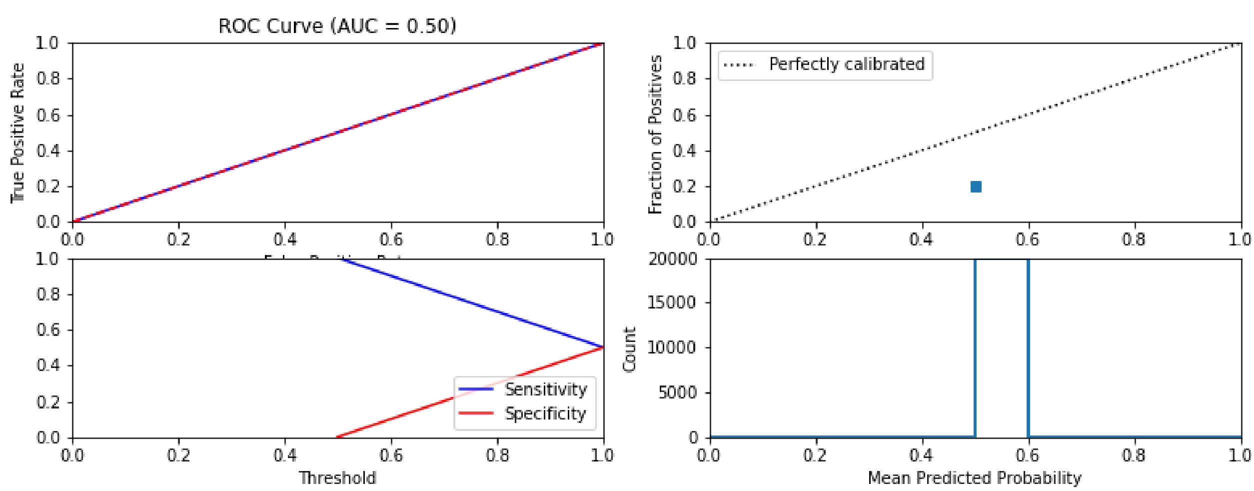
```
scores=[]
for i in range(100):
    data_dict = prepare_data(feature_subset = final_features,random_state=i)
    X_train_i=data_dict["X_train"]
    y_train_i=data_dict["y_train"]
    X_test_i=data_dict["X_test"]
    y_test_i=data_dict["y_test"]
    # GIVE THE BEST HYPERPARAMETER FROM THE PREVIOUS OUTPUT
    model = MLPClassifier(activation = 'logistic', alpha= 0.01, hidden_layer_sizes= (100,), 
    model.fit(X_train_i,y_train_i)
    scores.append(roc_auc_score(y_test_i,model.predict_proba(X_test_i)[:,1]))

print("Average test accuracy for MLP for 100 different splits :",np.mean(scores))
```

```
Average test accuracy for MLP for 100 different splits : 0.6932131560374222
```

Train and Test logistic regression model with features derived by LendingClub

In [24]:
```
## Find a LendingClub-defined feature and train a l1-regularized logistic regression model or
a_lendingclub_feature = 'grade'
data_dict = prepare_data(feature_subset = a_lendingclub_feature)
lc1_only_logistic =  LogisticRegression(penalty='l1', solver='liblinear')
cv_parameters=  {'C': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10]}
lc1_only_logistic = fit_classification(model=lc1_only_logistic,
                                data_dict=data_dict,
                                cv_parameters=cv_parameters,
                                model_name='Logistic Regression L1-regularized ',
                                random_state=default_seed,
                                output_to_file=True,
                                print_to_screen=True)
```

```
=========================================================
  Model: Logistic Regression L1-regularized
=========================================================
Fit time: 2.04 seconds
Optimal parameters:
{'C': 0.0001}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```



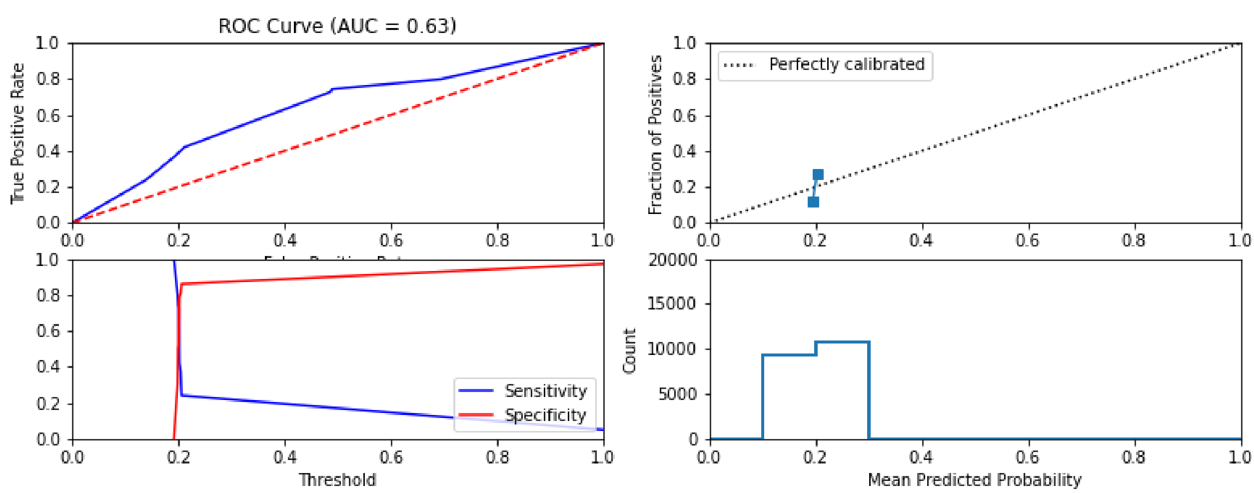

```
Similarity to LC grade ranking:  nan
Brier score: 0.25
Were parameters on edge? : True
Score variations around CV search grid : 0.03328617791461729
[0.80113333 0.80113333 0.80113333 0.80113333 0.80113333 0.80113333
 0.80113333 0.80086667 0.80086667 0.8009     0.8009    ]
```

In [25]:
```
data_dict['X_train'].shape
```

Out[25]:
```
(30000, 7)
```

In [26]:
```
## train a L2-regularized logistic regression model on data with only that feature
lc2_only_logistic = LogisticRegression(penalty='l2', solver='lbfgs')
cv_parameters = {'C': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10]}
l2_logistic = fit_classification(model=lc2_only_logistic,
                                 data_dict=data_dict,
                                 cv_parameters=cv_parameters,
                                 model_name='Logistic Regression L2-regularized',
                                 random_state=default_seed,
                                 output_to_file=True,
                                 print_to_screen=True)
```

```
==========================================================
  Model: Logistic Regression L2-regularized
==========================================================
Fit time: 2.36 seconds
Optimal parameters:
{'C': 0.0001}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```




```
Similarity to LC grade ranking:  0.642416265805365
Brier score: 0.15864697837477615
Were parameters on edge? : True
Score variations around CV search grid : 0.03328617791461729
[0.80113333 0.80113333 0.80113333 0.80113333 0.80113333 0.80113333
 0.80113333 0.80113333 0.80086667 0.80086667 0.8009    ]
```

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

In [27]:
```
data.head()

```

Out[27]:

	id	loan_amnt	funded_amnt	term	int_rate	grade	emp_length	home_ownership	annual_inc	verification_s
0	164190449	15000.0	15000.0	60 months	23.05	D	7 years	MORTGAGE	63800.0	Source V
1	163396715	12000.0	12000.0	36 months	7.56	A	10+ years	RENT	80000.0	Not V
2	164160069	18825.0	18825.0	36 months	6.46	A	< 1 year	OWN	95000.0	Source V
3	164174325	9075.0	9075.0	36 months	6.46	A	5 years	MORTGAGE	94000.0	Not V
4	164107998	6000.0	6000.0	36 months	16.95	C	5 years	MORTGAGE	83200.0	Source V

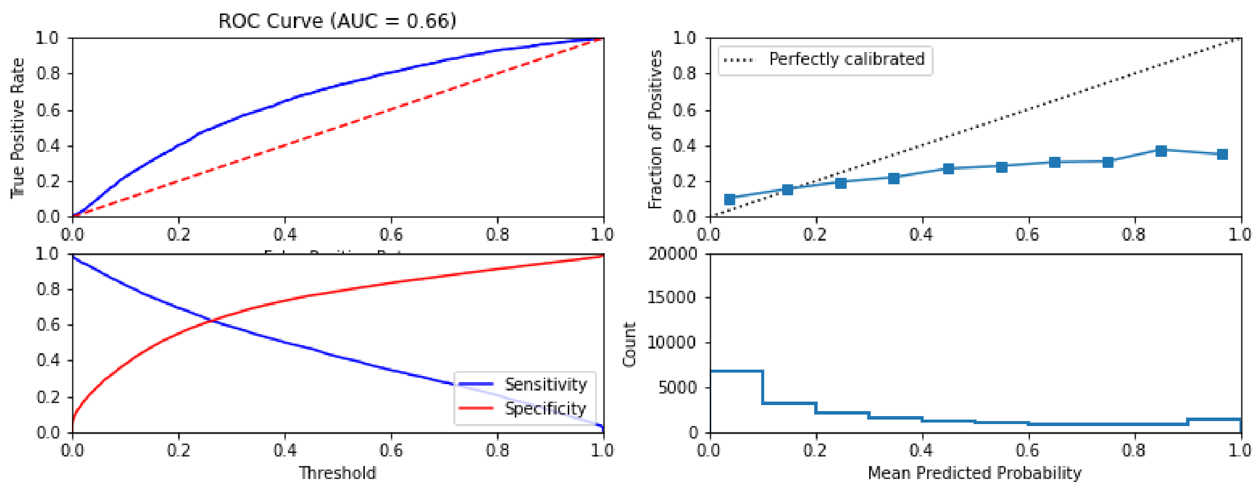
5 rows × 34 columns

In [73]:
```
## define your set of features to use in different models
final_features = [
'loan_amnt',
 'term',
 'int_rate',
 'emp_length',
 'home_ownership',
 'annual_inc',
 'verification_status',
 'purpose',
 'dti',
 'delinq_2yrs',
 'open_acc',
 'pub_rec',
 'fico_range_high',
 'fico_range_low',
 'revol_bal',
 'revol_util',
 'cr_hist']
# prepare the train, test data for training models
data_dict = prepare_data(feature_subset = final_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 final_features]
selected_features = all_features[idx]
selected_features.reset_index(drop=True,inplace=True)
```

In [29]:

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

gnb = GaussianNB()
gnb = fit_classification(model=gnb,
                         data_dict=data_dict,
                         cv_parameters={'var_smoothing': [1e-10, 1e-9, 1e-8, 1e-7, 1e-6]},
                         model_name='Gaussian Naive Bayes',
                         random_state=default_seed,
                         output_to_file=True,
                         print_to_screen=True)


```

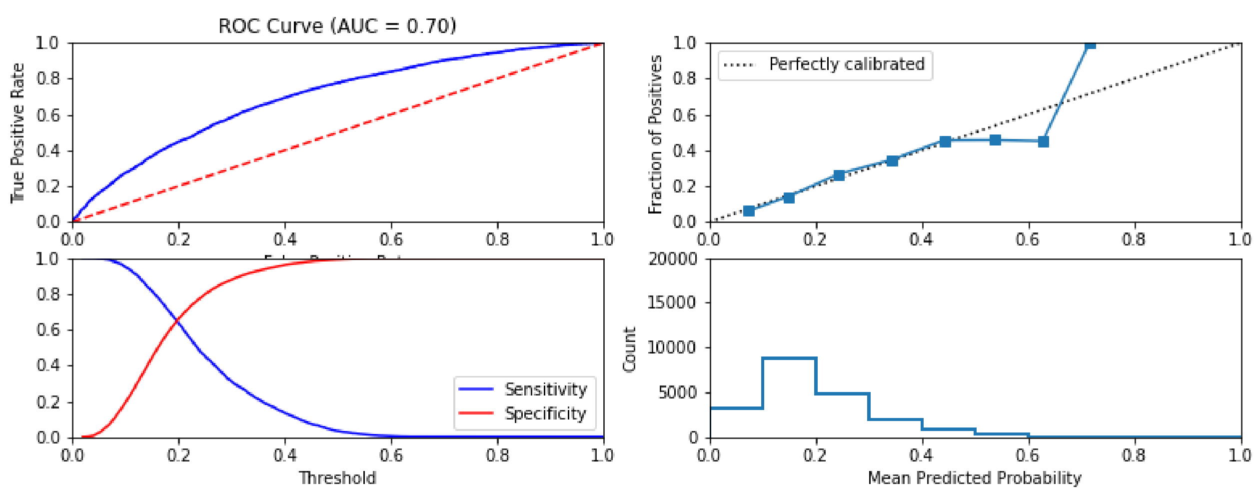
```
=========================================================
  Model: Gaussian Naive Bayes
=========================================================
Fit time: 0.93 seconds
Optimal parameters:
{'var_smoothing': 1e-06}

Accuracy-maximizing threshold was: 0.9999999999996945
Accuracy:  0.80095
              precision    recall  f1-score   support

  No default     0.8012    0.9996    0.8895     16024
     Default     0.2222    0.0005    0.0010      3976

    accuracy                         0.8010     20000
   macro avg     0.5117    0.5000    0.4452     20000
weighted avg     0.6861    0.8010    0.7128     20000
```




```
Similarity to LC grade ranking:  0.5387200456945855
Brier score: 0.21007466865929128
Were parameters on edge? : True
Score variations around CV search grid : 0.004733055660734049
[0.70423333 0.70423333 0.70423333 0.70423333 0.70426667]
```

In [30]:
```
## Train and test a l_1 regularized logistic regression classifier
l1_logistic =  LogisticRegression(penalty='l1', solver='liblinear')
cv_parameters = cv_parameters = {'C': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1

l1_logistic = fit_classification(model=l1_logistic,
                                data_dict=data_dict,
                                cv_parameters=cv_parameters,
                                model_name='Logistic Regression L1-regularized ',
                                random_state=default_seed,
                                output_to_file=True,
                                print_to_screen=True)
```

```
=========================================================
  Model: Logistic Regression L1-regularized
=========================================================
Fit time: 58.59 seconds
Optimal parameters:
{'C': 0.05}

Accuracy-maximizing threshold was: 0.48502185747196513
Accuracy:  0.80035
              precision    recall  f1-score   support

  No default     0.8064    0.9881    0.8880     16024
     Default     0.4767    0.0438    0.0802      3976

    accuracy                         0.8004     20000
   macro avg     0.6415    0.5159    0.4841     20000
weighted avg     0.7408    0.8004    0.7274     20000
```

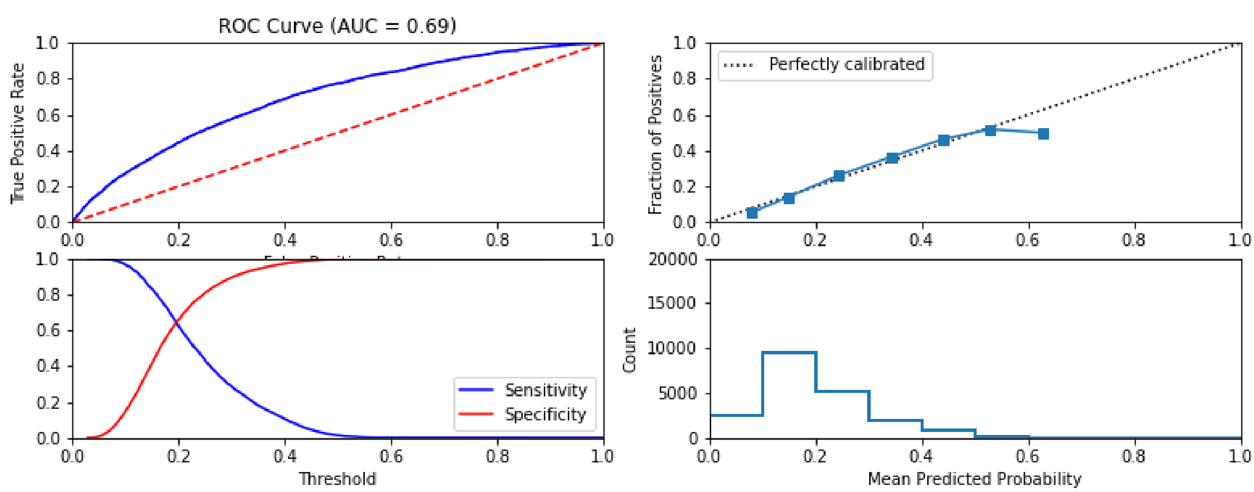



```
Similarity to LC grade ranking:  0.6610265055220242
Brier score: 0.14699498174963066
Were parameters on edge? : False
Score variations around CV search grid : 0.1745708466686148
[0.80113333 0.80113333 0.80113333 0.80113333 0.80056667 0.80196667
 0.80173333 0.8015     0.8014     0.8012     0.80116667]
```

In [31]:

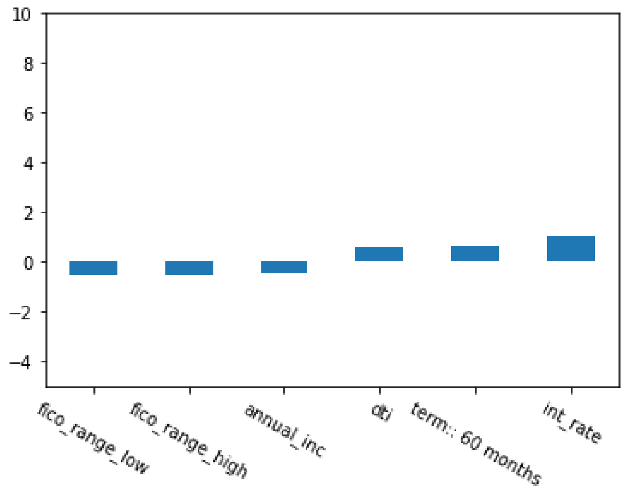
```
## Train and test a l_2 regularized logistic regression classifier

l2_logistic =  LogisticRegression(penalty='l2', solver='lbfgs')
cv_parameters = {'C': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10]}

l2_logistic = fit_classification(model=l2_logistic,
                                 data_dict=data_dict,
                                 cv_parameters=cv_parameters,
                                 model_name='Logistic Regression L2-regularized',
                                 random_state=default_seed,
                                 output_to_file=True,
                                 print_to_screen=True)
```

```
=========================================================
  Model: Logistic Regression L2-regularized
=========================================================
Fit time: 10.09 seconds
Optimal parameters:
{'C': 0.01}

Accuracy-maximizing threshold was: 0.45643224224179674
Accuracy:  0.80125
              precision    recall  f1-score   support

  No default     0.8061    0.9901    0.8887     16024
     Default     0.5016    0.0402    0.0745      3976

    accuracy                         0.8013     20000
   macro avg     0.6538    0.5152    0.4816     20000
weighted avg     0.7456    0.8013    0.7268     20000
```



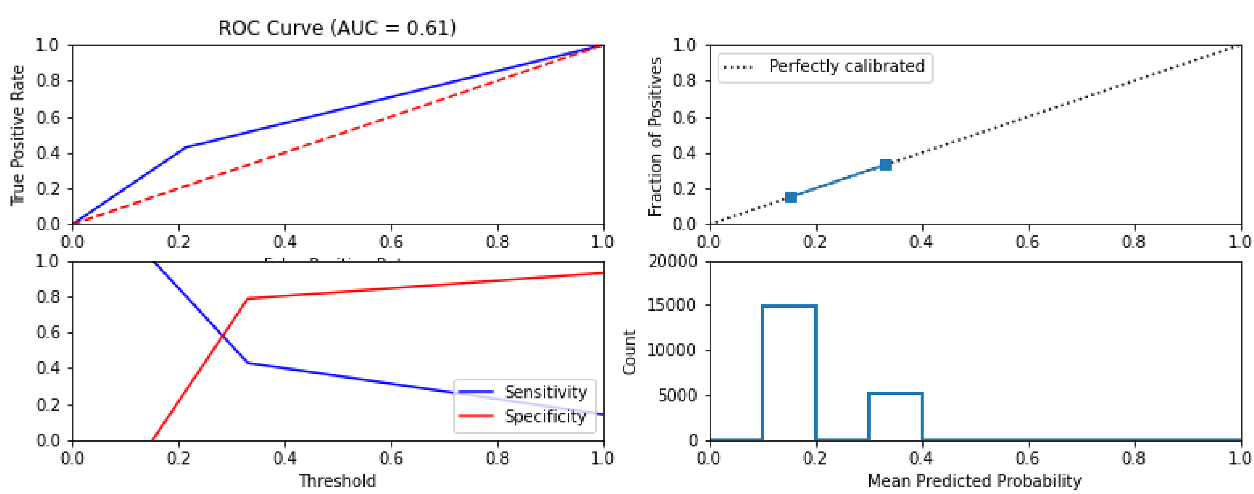

```
Similarity to LC grade ranking:  0.619306352648102
Brier score: 0.14733065040814727
Were parameters on edge? : False
Score variations around CV search grid : 0.11636605435958597
[0.80113333 0.80113333 0.80113333 0.8012     0.80206667 0.80196667
 0.802      0.8014     0.8014     0.80123333 0.80126667]
```

In [32]:
```
## 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_
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()
```



In [33]:
```
## Train and test a decision tree classifier

decision_tree = DecisionTreeClassifier(random_state=default_seed)
cv_parameters = cv_parameters = {
    'max_depth': [None, 5, 10, 15, 20, 25, 30, 35, 40],
    'min_samples_split': [2, 10, 20, 30, 40, 50, 60, 70, 80,90,100],
    'max_features': [None, 'sqrt', 'log2', 0.1, 0.25, 0.5, 0.75],
    'min_impurity_decrease': [0.0, 0.005, 0.01, 0.025, 0.05, 0.075, 0.1],
}

decision_tree = fit_classification(model=decision_tree,
                                    data_dict=data_dict,
                                    cv_parameters=cv_parameters,
                                    model_name='Decision Tree Classifier',
                                    random_state=default_seed,
                                    output_to_file=True,
                                    print_to_screen=True)
```

```
==========================================================
  Model: Decision Tree Classifier
==========================================================
Fit time: 1172.51 seconds
Optimal parameters:
{'max_depth': None, 'max_features': None, 'min_impurity_decrease': 0.005, 'min_samples_split':
2}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```

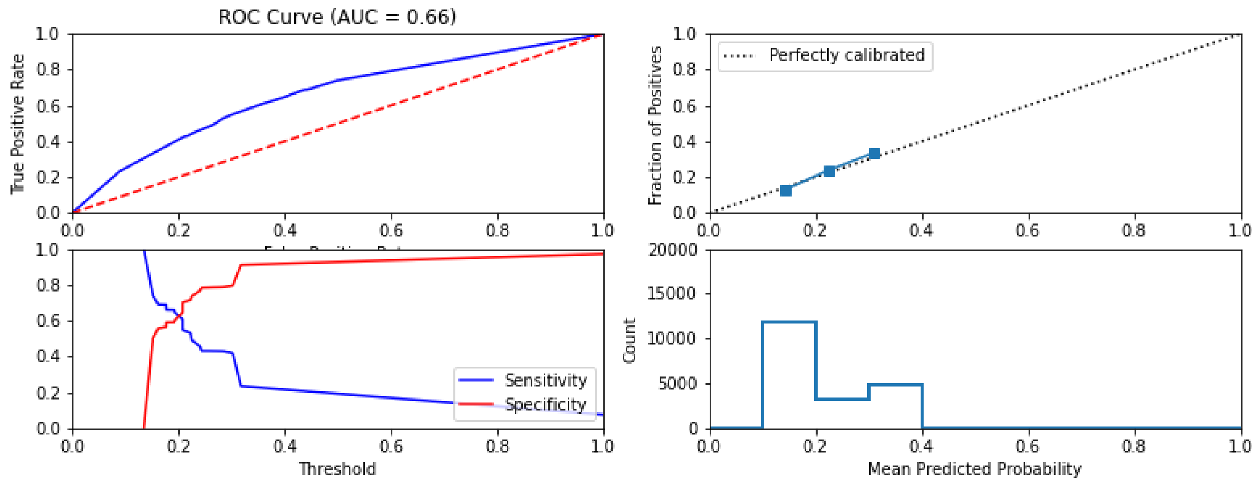



```
Similarity to LC grade ranking:  0.6768434354089615
Brier score: 0.153135354125818
Were parameters on edge? : True
Score variations around CV search grid : 13.501705916618121
[0.69606667 0.70873333 0.72493333 ... 0.80113333 0.80113333 0.80113333]
```

In [55]:

```
## Train and test a random forest classifier

random_forest = RandomForestClassifier(random_state=default_seed)
# cv_parameters = {
#     'n_estimators' : [10,25,50,75,100,150,200,300],
#     'bootstrap' :[True],
#     'max_depth': [None, 5, 10, 15, 20, 25, 30, 35, 40],
#     'min_samples_split': [2, 10, 20, 30, 40, 50, 60, 70, 80,90,100],
#     'max_features': [None, 'sqrt', 'log2', 0.1, 0.25, 0.5, 0.75],
#     'min_impurity_decrease': [0.0, 0.005, 0.01, 0.025, 0.05, 0.075, 0.1],
#     'max_samples': [None, 0.1, 0.25, 0.5, 0.75,1.0]}

cv_parameters = {
    'n_estimators' : [10,50,100,150],
    'bootstrap' :[True],
    'max_depth': [None],
    'min_samples_split': [2],
    'max_features': [None],
    'min_impurity_decrease': [0.005],
    'max_samples': [0.25]
}

random_forest = fit_classification(model=random_forest,
                                   data_dict=data_dict,
                                   cv_parameters=cv_parameters,
                                   model_name='Random Forest',
                                   random_state=default_seed,
                                   output_to_file=True,
                                   print_to_screen=True)
```

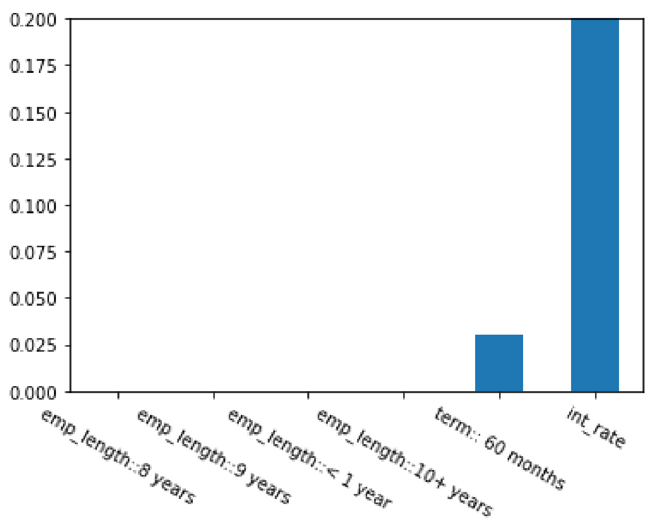
```
=========================================================
  Model: Random Forest
=========================================================
Fit time: 27.8 seconds
Optimal parameters:
{'bootstrap': True, 'max_depth': None, 'max_features': None, 'max_samples': 0.25, 'min_impurity
_decrease': 0.005, 'min_samples_split': 2, 'n_estimators': 10}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```




```
Similarity to LC grade ranking:  0.8232946413189874
Brier score: 0.15139641231303338
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.80113333 0.80113333 0.80113333 0.80113333]
```

In [35]:

```python
## 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()
```



Time stability test of YOURMODEL

In [36]:

```python
data.groupby(pd.to_datetime(data['issue_d']).dt.year)['id'].count()[-3:].sum()/data.groupby(
```

Out[36]: 0.24988903027254164

In [37]:

```python
data.groupby(pd.to_datetime(data['issue_d']).dt.year)['id'].count()[-3:].sum()
```

Out[37]: 270224

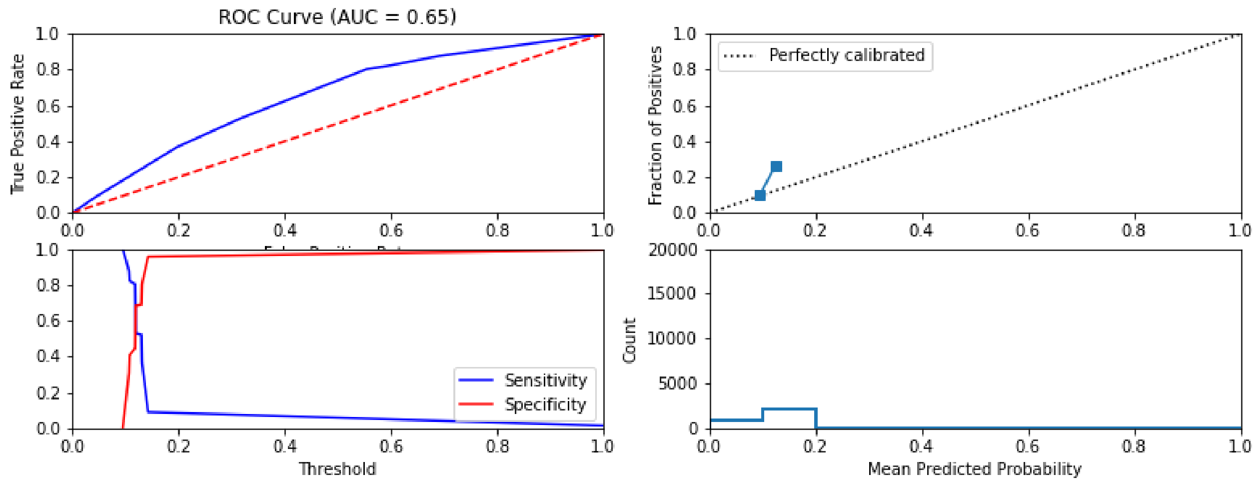
In [38]:
```
data["issue_d"].dtype
```

Out[38]: dtype('O')

In [77]:

```
## Define the time window of your train and test data

start_date_train = datetime.datetime.strptime("2010-01-01" , '%Y-%m-%d').date()
end_date_train =  datetime.datetime.strptime("2010-12-31" , '%Y-%m-%d').date()
start_date_test =  datetime.datetime.strptime("2018-01-01" , '%Y-%m-%d').date()
end_date_test = datetime.datetime.strptime("2018-12-31" , '%Y-%m-%d').date()

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 = 3000, feature_subset = fina


random_forest = RandomForestClassifier(random_state=default_seed)

cv_parameters = {
    'n_estimators' : [10,50,100,150],
    'bootstrap' :[True],
    'max_depth': [None],
    'min_samples_split': [2],
    'max_features': [None],
    'min_impurity_decrease': [0.005],
    'max_samples': [0.25]
}

random_forest = fit_classification(model=random_forest,
                                   data_dict=data_dict_test,
                                   cv_parameters=cv_parameters,
                                   model_name='Random Forest',
                                   random_state=default_seed,
                                   output_to_file=True,
                                   print_to_screen=True)


```

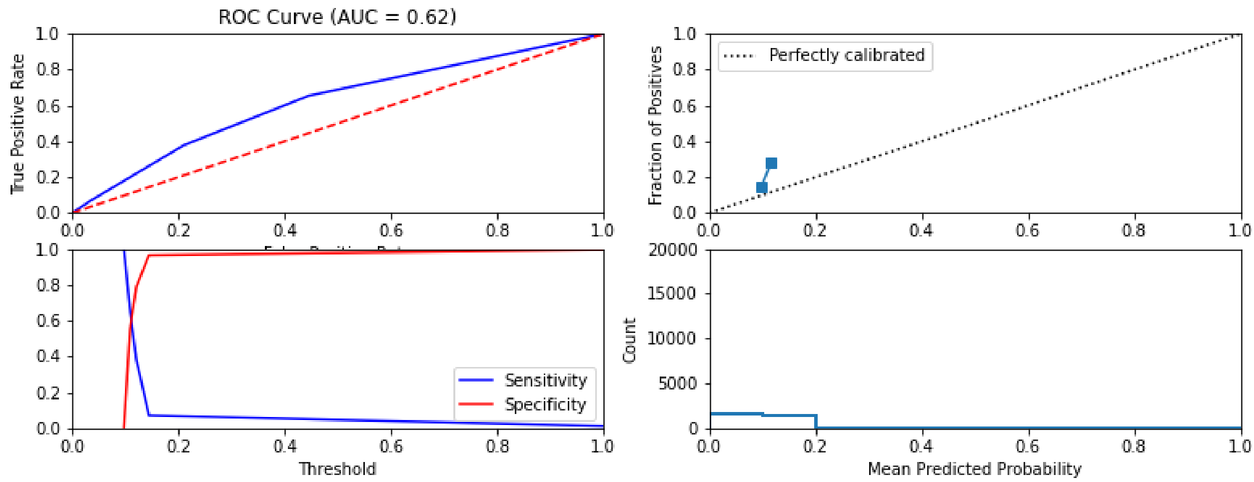
```
==========================================================
  Model: Random Forest
==========================================================
Fit time: 3.2 seconds
Optimal parameters:
{'bootstrap': True, 'max_depth': None, 'max_features': None, 'max_samples': 0.25, 'min_impurity
_decrease': 0.005, 'min_samples_split': 2, 'n_estimators': 10}

Accuracy-maximizing threshold was: 1
Accuracy:  0.7823333333333333
              precision    recall  f1-score   support

  No default     0.7823    1.0000    0.8779      2347
     Default     0.0000    0.0000    0.0000       653

    accuracy                         0.7823      3000
   macro avg     0.3912    0.5000    0.4389      3000
weighted avg     0.6120    0.7823    0.6868      3000
```




```
Similarity to LC grade ranking:  0.5948643136441721
Brier score: 0.1780745452709019
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.88414286 0.88414286 0.88414286 0.88414286]
```

In [78]:
```
## Define the time window of your train and test data

start_date_train = datetime.datetime.strptime("2017-01-01" , '%Y-%m-%d').date()
end_date_train =  datetime.datetime.strptime("2017-12-31" , '%Y-%m-%d').date()
start_date_test =  datetime.datetime.strptime("2018-01-01" , '%Y-%m-%d').date()
end_date_test = datetime.datetime.strptime("2018-12-31" , '%Y-%m-%d').date()

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 = 3000, feature_subset = fin


random_forest = RandomForestClassifier(random_state=default_seed)

cv_parameters = {
    'n_estimators' : [10,50,100,150],
    'bootstrap' :[True],
    'max_depth': [None],
    'min_samples_split': [2],
    'max_features': [None],
    'min_impurity_decrease': [0.005],
    'max_samples': [0.25]
}

random_forest = fit_classification(model=random_forest,
                                   data_dict=data_dict_test,
                                   cv_parameters=cv_parameters,
                                   model_name='Random Forest',
                                   random_state=default_seed,
                                   output_to_file=True,
                                   print_to_screen=True)


```

```
==========================================================
  Model: Random Forest
==========================================================
Fit time: 4.06 seconds
Optimal parameters:
{'bootstrap': True, 'max_depth': None, 'max_features': None, 'max_samples': 0.25, 'min_impurity
_decrease': 0.005, 'min_samples_split': 2, 'n_estimators': 10}

Accuracy-maximizing threshold was: 1
Accuracy:  0.7866666666666666
              precision    recall  f1-score   support

  No default     0.7867    1.0000    0.8806      2360
     Default     0.0000    0.0000    0.0000       640

    accuracy                         0.7867      3000
   macro avg     0.3933    0.5000    0.4403      3000
weighted avg     0.6188    0.7867    0.6927      3000
```

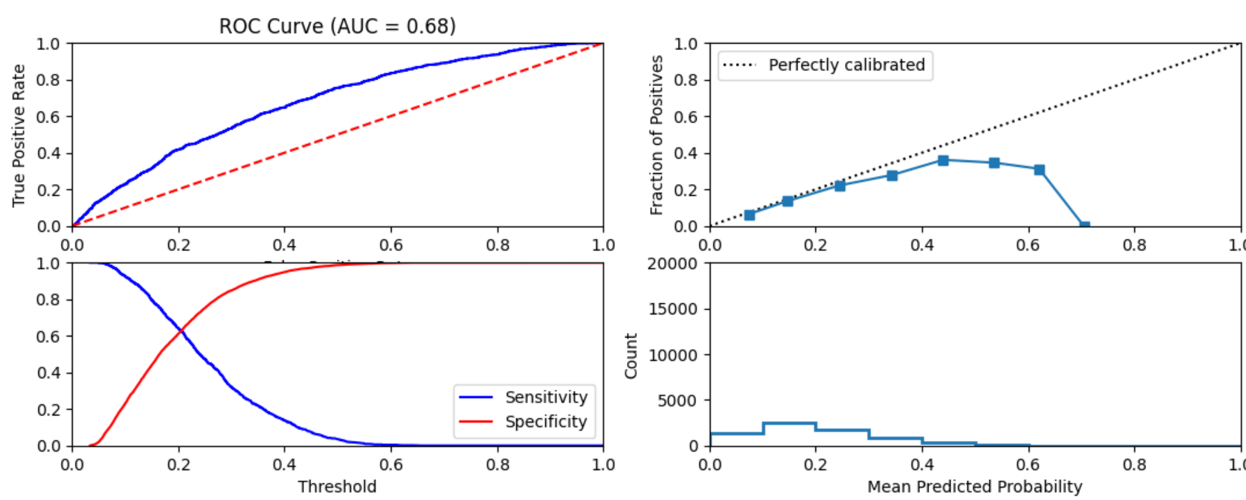



```
Similarity to LC grade ranking:  0.9149878324543382
Brier score: 0.17750102115477823
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.88742857 0.88742857 0.88742857 0.88742857]
```

In [74]:

```python
import xgboost as xgb

# Create an XGB classifier object
xgb_classifier = xgb.XGBClassifier(random_state=default_seed)

# Define a dictionary containing the cross-validation parameters
cv_parameters = {
    'learning_rate': [0.01, 0.05, 0.1],
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 5, 7, 9],
    'min_child_weight': [1, 3, 5, 7],
    'gamma': [0.0, 0.1, 0.2, 0.3],
    'subsample': [0.5, 0.5],
    'colsample_bytree': [0.5, 0.75, 1.0],
}

# Call the fit_classification() function with the appropriate parameters
xgb_model = fit_classification(model=xgb_classifier,
                               data_dict=data_dict_test,
                               cv_parameters=cv_parameters,
                               model_name='XGBoost',
                               random_state=default_seed,
                               output_to_file=True,
                               print_to_screen=True)

```

```
==========================================================
  Model: XGBoost
==========================================================
Fit time: 19288.62 seconds
Optimal parameters:
{'colsample_bytree': 0.75, 'gamma': 0.0, 'learning_rate': 0.1, 'max_depth': 3, 'min_child_weigh
t': 1, 'n_estimators': 50, 'subsample': 0.5}

Accuracy-maximizing threshold was: 0.4186105
Accuracy:  0.8124285714285714
              precision    recall  f1-score   support

  No default     0.8370    0.9599    0.8942      5781
     Default     0.3730    0.1132    0.1737      1219

    accuracy                         0.8124      7000
   macro avg     0.6050    0.5365    0.5340      7000
weighted avg     0.7562    0.8124    0.7687      7000
```

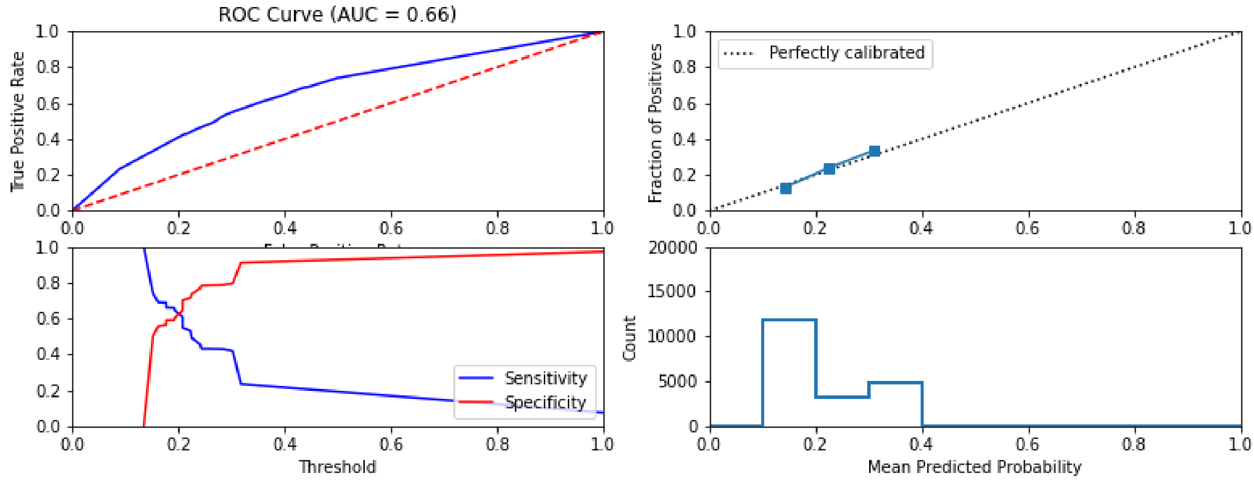


```
Similarity to LC grade ranking:  0.6616699391573095
Brier score: 0.13783851706505706
Were parameters on edge? : True
Score variations around CV search grid : 2.7743098904147496
[0.79988889 0.79988889 0.79988889 ... 0.78555556 0.78144444 0.78144444]
```

Train and test YOURMODEL on the original data

In [52]:
```
all_features = ['id', 'loan_amnt', 'funded_amnt', 'funded_amnt_inv', 'term', 'int_rate',
'installment', 'grade', 'sub_grade', 'emp_title', 'emp_length', 'home_ownership','annual_inc
'open_acc', 'pub_rec', 'last_pymnt_d', 'last_pymnt_amnt', 'fico_range_high',
'fico_range_low', 'last_fico_range_high', 'last_fico_range_low',
'application_type', 'revol_bal', 'revol_util', 'recoveries']
data_dict = prepare_data(feature_subset = all_features)

## Train and test YOURMODEL using this data

# WILL PROBABLY NEED TO CHANGE THIS MODEL
rf = RandomForestClassifier(random_state=default_seed)

cvparamList = {
    'n_estimators' : [10,50,100,150],
    'bootstrap' :[True],
    'max_depth': [None],
    'min_samples_split': [2],
    'max_features': [None],
    'min_impurity_decrease': [0.005],
    'max_samples': [0.25]
}

fit_classification(model=rf,data_dict=data_dict,
                                   cv_parameters=cvparamList,
                                   model_name='Random Forest',
                                   random_state=default_seed,
                                   output_to_file=True,
                                   print_to_screen=True)

# cvparamList = [x*2 for x in range(data.shape[1])]
# cvparamList.append(None)
# cv_parameters = {'max_depth': cvparamList}

# dt = DecisionTreeClassifier()
# fit_classification(dt,data_dict, cv_parameters,model_name='Decision Tree')

```

```
=========================================================
  Model: Random Forest
=========================================================
Fit time: 26.68 seconds
Optimal parameters:
{'bootstrap': True, 'max_depth': None, 'max_features': None, 'max_samples': 0.25, 'min_impurity
_decrease': 0.005, 'min_samples_split': 2, 'n_estimators': 10}

Accuracy-maximizing threshold was: 1
Accuracy:  0.8012
              precision    recall  f1-score   support

  No default     0.8012    1.0000    0.8896     16024
     Default     0.0000    0.0000    0.0000      3976

    accuracy                         0.8012     20000
   macro avg     0.4006    0.5000    0.4448     20000
weighted avg     0.6419    0.8012    0.7128     20000
```




```
Similarity to LC grade ranking:  0.8232946413189874
Brier score: 0.15139641231303338
Were parameters on edge? : True
Score variations around CV search grid : 0.0
[0.80113333 0.80113333 0.80113333 0.80113333]
```

Out[52]:
```
{'model': RandomForestClassifier(max_features=None, max_samples=0.25,
                        min_impurity_decrease=0.005, n_estimators=10,
                        random_state=1),
 'y_pred_labels': array([False, False, False, ..., False, False, False]),
 'y_pred_probs': array([0.30268909, 0.13469629, 0.19295033, ..., 0.19295033, 0.13469629,
        0.17679798])}
```

Test regression models

In [40]:
```
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 separa
      - predicted_default_return: prediction result for defaulted loans (valid if separate =
      - 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:
        print(ret_col)

        # print(data.loc[filter_train, ret_col])

        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']

            # 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_regu
                      'predicted_default_return':predicted_default_return,'r2_scores':test_s

        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_sco

        # 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_
                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_vari
                if print_to_screen:
                    print("Score variations around CV search grid (" + i + ") : " + str(scor

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

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

    return out
```

In [41]:
```
data_dict["X_train"].shape
```

Out[41]: (30000, 48)

l_1 regularized linear regression

In [42]:
```
## First, trying l1 regularized linear regression with hyper-parameters

from sklearn.linear_model import Lasso
cv_parameters = {
    'alpha': np.logspace(-4, 1, 6)
}

lasso_model = Lasso()

model_name = "Lasso Regression"

# Call the fit_regression function
reg_lasso = fit_regression(model=lasso_model, data_dict=data_dict, cv_parameters=cv_paramete
```

```
ret_PESS
=========================================================
  Model: Lasso Regression  Return column: ret_PESS
=========================================================
Fit time: 26.89 seconds
Optimal parameters:
{'alpha': 0.0001}

Testing r2 score: 0.03943372727730354
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -240.98193046644155
All test scores : [-0.01159498 -0.02752269 -0.03953679 -0.03953679 -0.03953679 -0.03953679]
ret_OPT
=========================================================
  Model: Lasso Regression  Return column: ret_OPT
=========================================================
Fit time: 33.79 seconds
Optimal parameters:
{'alpha': 0.0001}

Testing r2 score: 0.022349139592873102
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 278.8535474657161
All test scores : [ 0.00580081 -0.00361266 -0.01037496 -0.01037496 -0.01037496 -0.01037496]
ret_INTa
=========================================================
  Model: Lasso Regression  Return column: ret_INTa
=========================================================
Fit time: 38.59 seconds
Optimal parameters:
{'alpha': 0.0001}

Testing r2 score: 0.04590374629906013
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 121.40909006301875
All test scores : [ 0.03190879  0.02914522  0.00026634 -0.00683138 -0.00683138 -0.00683138]
ret_INTb
=========================================================
  Model: Lasso Regression  Return column: ret_INTb
=========================================================
Fit time: 34.71 seconds
Optimal parameters:
{'alpha': 0.0001}

Testing r2 score: 0.04660606844157045
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 103.15994635032756
All test scores : [ 0.03953327  0.03910599  0.01932546 -0.00124923 -0.00124923 -0.00124923]
```

l_2 regularized linear regressor

In [43]:

```python
## trying l2 regularized linear regression with hyper-parameters
from sklearn.linear_model import Ridge
cv_parameters = {
    'alpha': np.logspace(-4, 1, 6)
}

ridge_model = Ridge()

model_name = "Ridge Regression"

# Call the fit_regression function
reg_ridge = fit_regression(model=ridge_model, data_dict=data_dict, cv_parameters=cv_parameter

```

```
ret_PESS
=========================================================
  Model: Ridge Regression  Return column: ret_PESS
=========================================================
Fit time: 0.69 seconds
Optimal parameters:
{'alpha': 10.0}

Testing r2 score: 0.04013832365653891
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -1.9044519811991347
All test scores : [-0.01133323 -0.01133535 -0.01133302 -0.01130862 -0.01120559 -0.01112351]
ret_OPT
=========================================================
  Model: Ridge Regression  Return column: ret_OPT
=========================================================
Fit time: 0.85 seconds
Optimal parameters:
{'alpha': 10.0}

Testing r2 score: 0.022942250222890204
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 4.296401349888964
All test scores : [0.00546225 0.00546875 0.00547221 0.0054963  0.00561087 0.00570747]
ret_INTa
=========================================================
  Model: Ridge Regression  Return column: ret_INTa
=========================================================
Fit time: 0.77 seconds
Optimal parameters:
{'alpha': 10.0}

Testing r2 score: 0.046052520539923036
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 0.8906203451203547
All test scores : [0.0312151  0.03123445 0.03123895 0.03125764 0.03135088 0.03149561]
ret_INTb
=========================================================
  Model: Ridge Regression  Return column: ret_INTb
=========================================================
Fit time: 0.81 seconds
Optimal parameters:
{'alpha': 10.0}

Testing r2 score: 0.046418691859571815
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 0.7625539462892043
All test scores : [0.03903546 0.03907207 0.03907843 0.03909456 0.03918049 0.03933542]
```

Multi-layer perceptron regressor

In [44]:
```
## trying multi-layer perceptron regression with hyper-parameters

from sklearn.neural_network import MLPRegressor

mlp_model = MLPRegressor(random_state=default_seed,solver='adam',learning_rate='constant')

# Define the hyperparameters to optimize using cross-validation
cv_parameters = {
    'hidden_layer_sizes': [(10,), (50,50)],
    'activation': ['logistic'],
    'learning_rate_init': [0.001, 0.01, 0.1]
    }


reg_mlp = fit_regression(model = mlp_model, data_dict= data_dict,
                      cv_parameters = cv_parameters,
                      separate = False,
                      model_name = "MLP Regressor",
                      random_state = default_seed,
                      output_to_file = True,
                      print_to_screen = True)
```

```
ret_PESS
=========================================================
  Model: MLP Regressor  Return column: ret_PESS
=========================================================
Fit time: 58.16 seconds
Optimal parameters:
{'activation': 'logistic', 'hidden_layer_sizes': (50, 50), 'learning_rate_init': 0.001}

Testing r2 score: 0.03949738069747355
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 945.7076234038603
All test scores : [-0.02532798 -0.03499739 -0.00564384  0.00860272 -0.07275383 -0.00154256]
ret_OPT
=========================================================
  Model: MLP Regressor  Return column: ret_OPT
=========================================================
Fit time: 60.17 seconds
Optimal parameters:
{'activation': 'logistic', 'hidden_layer_sizes': (50, 50), 'learning_rate_init': 0.001}

Testing r2 score: 0.010488550370333027
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 527.8857744491416
All test scores : [-0.00259105 -0.02542251 -0.00890724  0.00594143 -0.02008125 -0.00092688]
ret_INTa
=========================================================
  Model: MLP Regressor  Return column: ret_INTa
=========================================================
Fit time: 74.35 seconds
Optimal parameters:
{'activation': 'logistic', 'hidden_layer_sizes': (50, 50), 'learning_rate_init': 0.01}

Testing r2 score: 0.03726765752265204
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 122.29856460287614
All test scores : [ 0.03232255 -0.00908212  0.02848989  0.02835864  0.04072962  0.03318835]
ret_INTb
=========================================================
  Model: MLP Regressor  Return column: ret_INTb
=========================================================
Fit time: 550.08 seconds
Optimal parameters:
{'activation': 'logistic', 'hidden_layer_sizes': (10,), 'learning_rate_init': 0.001}

Testing r2 score: 0.045763288763081444
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 540.514761583225
All test scores : [ 0.03853431  0.03621658  0.02062731  0.03430834 -0.16974932  0.01574494]
```

Random forest regressor

In [45]:
```
## trying random forest regression with hyper-parameters

from sklearn.ensemble import RandomForestRegressor

reg_rf = RandomForestRegressor(random_state=default_seed)

# Define hyperparameters to be tuned
cv_parameters = {
    'n_estimators': [10,50,100,150],
    'max_depth': [None],
    'min_samples_split': [2],
    'min_samples_leaf': [1],
    'max_features': [None]
}

model_name = "Random Forest Regressor"

reg_rf = fit_regression(model = reg_rf, data_dict= data_dict,
                      cv_parameters = cv_parameters,
                      separate = False,
                      model_name = model_name,
                      random_state = default_seed,
                      output_to_file = True,
                      print_to_screen = True )
```

```
ret_PESS
=========================================================
  Model: Random Forest Regressor  Return column: ret_PESS
=========================================================
Fit time: 673.23 seconds
Optimal parameters:
{'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_est
imators': 150}

Testing r2 score: 0.0343663701807676
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -361.575116623546
All test scores : [-0.12749952 -0.04419631 -0.03229759 -0.0276227 ]
ret_OPT
=========================================================
  Model: Random Forest Regressor  Return column: ret_OPT
=========================================================
Fit time: 668.95 seconds
Optimal parameters:
{'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_est
imators': 150}

Testing r2 score: 0.006839979119483308
Were parameters on edge (model) : True
Score variations around CV search grid (model) : -663.9540423810339
All test scores : [-0.12699169 -0.03704576 -0.01941059 -0.01662295]
ret_INTa
=========================================================
  Model: Random Forest Regressor  Return column: ret_INTa
=========================================================
Fit time: 654.3 seconds
Optimal parameters:
{'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_est
imators': 150}

Testing r2 score: 0.028794389508437934
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 1051.2602309284296
All test scores : [-0.08775757 -0.00635503  0.00438341  0.0092254 ]
ret_INTb
=========================================================
  Model: Random Forest Regressor  Return column: ret_INTb
=========================================================
Fit time: 622.93 seconds
Optimal parameters:
{'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_est
imators': 150}

Testing r2 score: 0.03314962065316396
Were parameters on edge (model) : True
Score variations around CV search grid (model) : 516.5196666997973
All test scores : [-0.08813538  0.00295032  0.01769883  0.02115996]
```

Test investment strategies

Now we test several investment strategies using the learning models above

In [47]:

```
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 o
                   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 functi
      - 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 = num_loans
```

```
            ## Find the average return for these loans
            ret_test = data_test[ret_col].iloc[:num_loans].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(dro

            ## 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

            # 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_re
            data_test['predicted_default_return'] = regressor[ret_col]['predicted_default_re

            # 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_ret

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

            ## 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

            # 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 [49]:

```
## 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)
strat_rand = test_investments(data_dict=data_dict,classifier=None,regressor=None,
                              strategy = test_strategy,num_loans = 1000,random_state = defau
                        output_to_file = True)

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

```
strategy: Random
ret_PESS: -0.0033487895710861567
ret_OPT: 0.03696126237833046
ret_INTa: 0.4154103482654324
ret_INTb: 1.2638347895926758
```

In [56]:
```
test_strategy = 'Default-based'

print('strategy:',test_strategy)
strat_def = test_investments(data_dict=data_dict, classifier=random_forest,
                             regressor=None,strategy=test_strategy,num_loans = 1000, random_s
                        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: 0.01807470428073583
ret_OPT: 0.04363484942793374
ret_INTa: 0.41647524602330244
ret_INTb: 1.2569719525143639
```

In [53]:
```
test_strategy = 'Return-based'

print('strategy:',test_strategy)
strat_ret = test_investments(data_dict=data_dict,classifier=None,
                             regressor=reg_rf,strategy=test_strategy,num_loans = 1000,random_
                        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: 0.025822308646065658
ret_OPT: 0.03957188166939146
ret_INTa: 0.4145379947842627
ret_INTb: 1.2601514001855478
```

In [57]:

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

## For the Default-return-based strategy we need to fit a new regressor with separate = True
from sklearn.linear_model import Ridge
cv_parameters = {
    'n_estimators': [10,50,100,150],
    'max_depth': [None],
    'min_samples_split': [2],
    'min_samples_leaf': [1],
    'max_features': [None]
}

reg_rf_drb = RandomForestRegressor(random_state=default_seed)
model_name = "Random Forest Regressor"

reg_separate = fit_regression(model=reg_rf_drb, data_dict=data_dict, cv_parameters=cv_parame
                              separate = True, model_name=model_name)

print('strategy:',test_strategy)
strat_defret = test_investments(data_dict=data_dict, classifier=random_forest,
                             regressor=reg_separate, strategy=test_strategy, num_loans = 100
                        output_to_file = True)

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

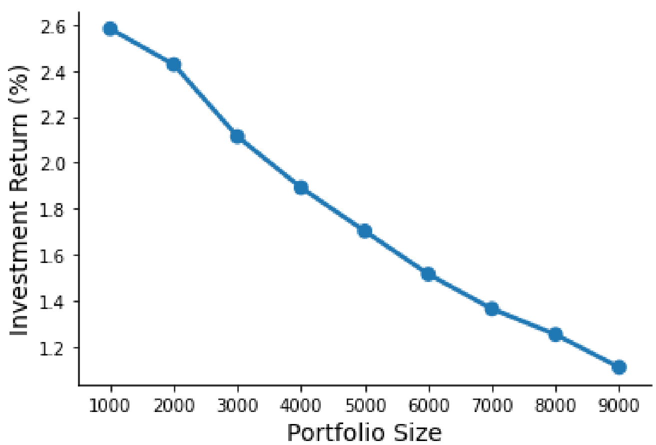
```
ret_PESS
=========================================================
  Model: Random Forest Regressor  Return column: ret_PESS
=========================================================
Fit time: 640.66 seconds
Optimal parameters:
model_0: {'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split':
2, 'n_estimators': 150} model_1 {'max_depth': None, 'max_features': None, 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 150}

Testing r2 scores:
model_0: 0.13051833883311303
model_1: 0.15007979823376671
Were parameters on edge (model_0) : True
Score variations around CV search grid (model_0) : -82.59233621296465
All test scores : [-0.22889597 -0.14026768 -0.13057324 -0.12535903]
Were parameters on edge (model_1) : True
Score variations around CV search grid (model_1) : -248.76349897216127
All test scores : [-0.11530262 -0.04303475 -0.03472609 -0.0330604 ]
ret_OPT
=========================================================
  Model: Random Forest Regressor  Return column: ret_OPT
=========================================================
Fit time: 637.38 seconds
Optimal parameters:
model_0: {'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split':
2, 'n_estimators': 150} model_1 {'max_depth': None, 'max_features': None, 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 150}

Testing r2 scores:
model_0: 0.1209768713776106
model_1: 0.13672690313375557
Were parameters on edge (model_0) : True
Score variations around CV search grid (model_0) : 112.43387125845724
All test scores : [-0.01355904  0.09077863  0.10456048  0.10904919]
Were parameters on edge (model_1) : True
Score variations around CV search grid (model_1) : -188.97092459598034
All test scores : [-0.15464783 -0.05874596 -0.05587698 -0.05351674]
ret_INTa
=========================================================
  Model: Random Forest Regressor  Return column: ret_INTa
=========================================================
Fit time: 676.31 seconds
Optimal parameters:
model_0: {'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split':
2, 'n_estimators': 150} model_1 {'max_depth': None, 'max_features': None, 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 150}

Testing r2 scores:
model_0: 0.01926628857494128
model_1: 0.0657836787502426
Were parameters on edge (model_0) : True
Score variations around CV search grid (model_0) : -503.2407037634867
All test scores : [-0.16983259 -0.04718882 -0.02912604 -0.02815337]
Were parameters on edge (model_1) : True
Score variations around CV search grid (model_1) : -106.6218910057883
All test scores : [-0.20976479 -0.11827711 -0.10651641 -0.10152109]
ret_INTb
=========================================================
  Model: Random Forest Regressor  Return column: ret_INTb
=========================================================
Fit time: 628.57 seconds
Optimal parameters:
model_0: {'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split':
2, 'n_estimators': 150} model_1 {'max_depth': None, 'max_features': None, 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 150}
```

```
Testing r2 scores:
model_0: 0.020893906581689103
model_1: 0.08337553445791257
Were parameters on edge (model_0) : True
Score variations around CV search grid (model_0) : 2069.0391262892945
All test scores : [-0.10259713 -0.01112183  0.00222889  0.00521052]
Were parameters on edge (model_1) : True
Score variations around CV search grid (model_1) : -139.11773354519755
All test scores : [-0.15848238 -0.08265055 -0.07064342 -0.06627797]
strategy: Default-return-based
ret_PESS: 0.02995912893746814
ret_OPT: 0.034327414198711825
ret_INTa: 0.41643379656353496
ret_INTb: 1.2521583045099454
```

Sensitivity test of portfolio size

In [58]:

```
## Test the best-performing data-driven strategy on different portfolio sizes

result_sensitivity = []

test_strategy = 'Return-based'
## 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_rf,strategy=test_strategy,num_loans = num_loans,r
                        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 [ ]:

```
```