Exploring Ensemble Methods

In this assignment, we will explore the use of boosting. We will use the pre-implemented gradient boosted trees in GraphLab Create. You will:

- Use SFrames to do some feature engineering.
- Train a boosted ensemble of decision-trees (gradient boosted trees) on the LendingClub dataset.
- Predict whether a loan will default along with prediction probabilities (on a validation set).
- Evaluate the trained model and compare it with a baseline.
- Find the most positive and negative loans using the learned model.
- Explore how the number of trees influences classification performance.

Let's get started!

Fire up Graphlab Create

In [1]:

```
import graphlab
```

Load LendingClub dataset

We will be using the LendingClub (https://www.lendingclub.com/) data. As discussed earlier, the LendingClub (https://www.lendingclub.com/) is a peer-to-peer leading company that directly connects borrowers and potential lenders/investors.

Just like we did in previous assignments, we will build a classification model to predict whether or not a loan provided by lending club is likely to default.

Let us start by loading the data.

In [2]:

```
loans = graphlab.SFrame('lending-club-data.gl/')
```

```
This non-commercial license of GraphLab Create for academic use is assigned
to amitha353@gmail.com and will expire on May 07, 2019.
```

```
[INFO] graphlab.cython.cy_server: GraphLab Create v2.1 started. Logging:
C:\Users\Amitha\AppData\Local\Temp\graphlab_server_1533053556.log.0
```

Let's quickly explore what the dataset looks like. First, let's print out the column names to see what features we have in this dataset. We have done this in previous assignments, so we won't belabor this here.

In [3]:

```
loans.column_names()
```

Out[3]:

```
['id',
 'member_id',
 'loan_amnt',
 'funded_amnt',
 'funded_amnt_inv',
 'term',
 'int_rate',
 'installment',
 'grade',
 'sub_grade',
 'emp_title',
 'emp_length',
 'home_ownership',
 'annual_inc',
 'is_inc_v',
 'issue_d',
 'loan_status',
 'pymnt_plan',
 'url',
 'desc',
 'purpose',
 'title',
 'zip_code',
 'addr_state',
 'dti',
 'delinq_2yrs',
 'earliest_cr_line',
 'inq_last_6mths',
 'mths_since_last_delinq',
 'mths_since_last_record',
 'open_acc',
 'pub_rec',
 'revol_bal',
 'revol_util',
 'total_acc',
 'initial_list_status',
 'out_prncp',
 'out_prncp_inv',
 'total_pymnt',
 'total_pymnt_inv',
 'total_rec_prncp',
 'total_rec_int',
 'total_rec_late_fee',
 'recoveries',
 'collection_recovery_fee',
 'last_pymnt_d',
 'last_pymnt_amnt',
 'next_pymnt_d',
 'last_credit_pull_d',
 'collections_12_mths_ex_med',
 'mths_since_last_major_derog',
 'policy_code',
 'not_compliant',
 'status',
 'inactive_loans',
 'bad_loans',
 'emp_length_num',
 'grade_num',
 'sub_grade_num',
 'delinq_2yrs_zero',
 'pub_rec_zero',
 'collections_12_mths_zero',
 'short_emp',
 'payment_inc_ratio',
 'final_d',
 'last_delinq_none',
 'last_record_none',
 'last_major_derog_none']
```

Modifying the target column

The target column (label column) of the dataset that we are interested in is called `bad_loans`. In this column **1** means a risky (bad) loan **0** means a safe loan.

As in past assignments, in order to make this more intuitive and consistent with the lectures, we reassign the target to be:

- **+1** as a safe loan,
- **-1** as a risky (bad) loan.

We put this in a new column called `safe_loans`.

In [4]:

```
loans['safe_loans'] = loans['bad_loans'].apply(lambda x : +1 if x==0 else -1)
loans = loans.remove_column('bad_loans')
```

Selecting features

In this assignment, we will be using a subset of features (categorical and numeric). The features we will be using are **described in the code comments** below. If you are a finance geek, the LendingClub (https://www.lendingclub.com/) website has a lot more details about these features.

The features we will be using are described in the code comments below:

In [5]:

```
target = 'safe_loans'
features = ['grade',                     # grade of the loan (categorical)
            'sub_grade_num',             # sub-grade of the loan as a number from 0 to 1
            'short_emp',                 # one year or less of employment
            'emp_length_num',            # number of years of employment
            'home_ownership',            # home_ownership status: own, mortgage or rent
            'dti',                       # debt to income ratio
            'purpose',                   # the purpose of the loan
            'payment_inc_ratio',         # ratio of the monthly payment to income
            'delinq_2yrs',               # number of delinquincies
            'delinq_2yrs_zero',          # no delinquincies in last 2 years
            'inq_last_6mths',            # number of creditor inquiries in last 6 months
            'last_delinq_none',          # has borrower had a delinquincy
            'last_major_derog_none',     # has borrower had 90 day or worse rating
            'open_acc',                  # number of open credit accounts
            'pub_rec',                   # number of derogatory public records
            'pub_rec_zero',              # no derogatory public records
            'revol_util',                # percent of available credit being used
            'total_rec_late_fee',        # total late fees received to day
            'int_rate',                  # interest rate of the loan
            'total_rec_int',             # interest received to date
            'annual_inc',                # annual income of borrower
            'funded_amnt',               # amount committed to the loan
            'funded_amnt_inv',           # amount committed by investors for the loan
            'installment',               # monthly payment owed by the borrower
           ]
```

Skipping observations with missing values

Recall from the lectures that one common approach to coping with missing values is to **skip** observations that contain missing values.

We run the following code to do so:

In [6]:

```
loans, loans_with_na = loans[[target] + features].dropna_split()

# Count the number of rows with missing data
num_rows_with_na = loans_with_na.num_rows()
num_rows = loans.num_rows()
print 'Dropping %s observations; keeping %s ' % (num_rows_with_na, num_rows)
```

```
Dropping 29 observations; keeping 122578
```

Fortunately, there are not too many missing values. We are retaining most of the data.

Make sure the classes are balanced

We saw in an earlier assignment that this dataset is also imbalanced. We will undersample the larger class (safe loans) in order to balance out our dataset. We used `seed=1` to make sure everyone gets the same results.

In [7]:

```
safe_loans_raw = loans[loans[target] == 1]
risky_loans_raw = loans[loans[target] == -1]

# Undersample the safe loans.
percentage = len(risky_loans_raw)/float(len(safe_loans_raw))
safe_loans = safe_loans_raw.sample(percentage, seed = 1)
risky_loans = risky_loans_raw
loans_data = risky_loans.append(safe_loans)

print "Percentage of safe loans                 :", len(safe_loans) / float(len(loans_data)
print "Percentage of risky loans                :", len(risky_loans) / float(len(loans_data
print "Total number of loans in our new dataset :", len(loans_data)
```

```
Percentage of safe loans                 : 0.502247166849
Percentage of risky loans                : 0.497752833151
Total number of loans in our new dataset : 46503
```

Checkpoint: You should now see that the dataset is balanced (approximately 50-50 safe vs risky loans).

Note: There are many approaches for dealing with imbalanced data, including some where we modify the learning approaches are beyond the scope of this course, but some of them are reviewed in this paper (http://ieeexplore.iee tp=&arnumber=5128907&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F69%2F5173046%2F05128907.pdf For this assignment, we use the simplest possible approach, where we subsample the overly represented class to dataset. In general, and especially when the data is highly imbalanced, we recommend using more advanced meth

Split data into training and validation sets

We split the data into training data and validation data. We used `seed=1` to make sure everyone gets the same results. We will use the validation data to help us select model parameters.

In [8]:

```
train_data, validation_data = loans_data.random_split(.8, seed=1)
```

Gradient boosted tree classifier

Gradient boosted trees are a powerful variant of boosting methods; they have been used to win many Kaggle (https://www.kaggle.com/) competitions, and have been widely used in industry. We will explore the predictive power of multiple decision trees as opposed to a single decision tree.

Additional reading: If you are interested in gradient boosted trees, here is some additional reading material:

- GraphLab Create user guide (https://dato.com/learn/userguide/supervised-learning/boosted_trees_classifier.html)
- Advanced material on boosted trees (http://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf)

We will now train models to predict `safe_loans` using the features above. In this section, we will experiment with training an ensemble of 5 trees. To cap the ensemble classifier at 5 trees, we call the function with **max_iterations=5** (recall that each iterations corresponds to adding a tree). We set `validation_set=None` to make sure everyone gets the same results.

In [9]:

```
model_5 = graphlab.boosted_trees_classifier.create(train_data, validation_set=None,
        target = target, features = features, max_iterations = 5)
```

```
Boosted trees classifier:

--------------------------------------------------------

Number of examples          : 37219

Number of classes           : 2

Number of feature columns   : 24

Number of unpacked features : 24

+-----------+--------------+-------------------+-------------------+

| Iteration | Elapsed Time | Training-accuracy | Training-log_loss |

+-----------+--------------+-------------------+-------------------+

| 1         | 0.130441     | 0.657541          | 0.657139          |

| 2         | 0.261367     | 0.656976          | 0.636157          |

| 3         | 0.402286     | 0.664983          | 0.623206          |

| 4         | 0.537117     | 0.668476          | 0.613783          |

| 5         | 0.674037     | 0.673339          | 0.606229          |

+-----------+--------------+-------------------+-------------------+
```

Making predictions

Just like we did in previous sections, let us consider a few positive and negative examples **from the validation set**. We will do the following:

- Predict whether or not a loan is likely to default.
- Predict the probability with which the loan is likely to default.

In [10]:

```
# Select all positive and negative examples.
validation_safe_loans = validation_data[validation_data[target] == 1]
validation_risky_loans = validation_data[validation_data[target] == -1]

# Select 2 examples from the validation set for positive & negative loans
sample_validation_data_risky = validation_risky_loans[0:2]
sample_validation_data_safe = validation_safe_loans[0:2]

# Append the 4 examples into a single dataset
sample_validation_data = sample_validation_data_safe.append(sample_validation_data_risky)
sample_validation_data
```

Out[10]:

safe_loans	grade	sub_grade_num	short_emp	emp_length_num	hon
1	B	0.2	0	3	M
1	B	0.6	1	1	
-1	D	0.4	0	3	
-1	A	1.0	0	11	M

payment_inc_ratio	delinq_2yrs	delinq_2yrs_zero	inq_last_6mths	last_d
6.30496	0	1	0	
13.4952	0	1	0	
2.96736	3	0	0	
1.90524	0	1	0	

pub_rec	pub_rec_zero	revol_util	total_rec_late_fee	int_rate	total_re
0	1	93.9	0.0	9.91	823.4
0	1	59.1	0.0	11.71	1622.
0	1	59.5	0.0	16.77	719.1
0	1	62.1	0.0	8.9	696.9

installment
483.38
281.15
123.65
158.77

[4 rows x 25 columns]

Predicting on sample validation data

For each row in the **sample_validation_data**, write code to make **model_5** predict whether or not the loan is classified as a **safe loan**.

Hint: Use the `predict` method in `model_5` for this.

In [12]:

```
model_5.predict(sample_validation_data)
```

Out[12]:

```
dtype: int
Rows: 4
[1L, 1L, -1L, 1L]
```

In [15]:

```
len(sample_validation_data[sample_validation_data['safe_loans'] == model_5.predict(sample_v
```

Out[15]:

```
0.75
```

Quiz Question: What percentage of the predictions on `sample_validation_data` did `model_5` get correct?

Prediction probabilities

For each row in the **sample_validation_data**, what is the probability (according **model_5**) of a loan being classified as **safe**?

Hint: Set `output_type='probability'` to make **probability** predictions using `model_5` on `sample_validation_data`:

In [17]:

```
predictProbability = model_5.predict(sample_validation_data, output_type='probability')
print predictProbability
```

```
[0.7045905590057373, 0.5963408946990967, 0.44925159215927124, 0.611909985542
2974]
```

In [26]:

```
print predictProbability.min()
```

```
0.449251592159
```

Quiz Question: According to **model_5**, which loan is the least likely to be a safe loan?

Checkpoint: Can you verify that for all the predictions with `probability >= 0.5`, the model predicted the label **+1**?

Evaluating the model on the validation data

Recall that the accuracy is defined as follows:

$$\text{accuracy} = \frac{\text{\# correctly classified examples}}{\text{\# total examples}}$$

Evaluate the accuracy of the **model_5** on the **validation_data**.

Hint: Use the `.evaluate()` method in the model.

In [27]:

```
model_5.evaluate(validation_data)
```

Out[27]:

```
{'accuracy': 0.66813873330461,
 'auc': 0.7247215702188436,
 'confusion_matrix': Columns:
 	target_label	int
 	predicted_label	int
 	count	int

 Rows: 4

 Data:
 +--------------+-----------------+-------+
 | target_label | predicted_label | count |
 +--------------+-----------------+-------+
 |      -1      |        1        |  1618 |
 |      -1      |        -1       |  3054 |
 |      1       |        -1       |  1463 |
 |      1       |        1        |  3149 |
 +--------------+-----------------+-------+
 [4 rows x 3 columns],
 'f1_score': 0.6715001599317625,
 'log_loss': 0.6176131769648966,
 'precision': 0.6605831760016782,
 'recall': 0.6827840416305291,
 'roc_curve': Columns:
 	threshold	float
 	fpr	float
 	tpr	float
 	p	int
 	n	int

 Rows: 100001

 Data:
 +-----------+-----+-----+------+------+
 | threshold | fpr | tpr |  p   |  n   |
 +-----------+-----+-----+------+------+
 |    0.0    | 1.0 | 1.0 | 4612 | 4672 |
 |   1e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   2e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   3e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   4e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   5e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   6e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   7e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   8e-05   | 1.0 | 1.0 | 4612 | 4672 |
 |   9e-05   | 1.0 | 1.0 | 4612 | 4672 |
 +-----------+-----+-----+------+------+
 [100001 rows x 5 columns]
 Note: Only the head of the SFrame is printed.
 You can use print_rows(num_rows=m, num_columns=n) to print more rows and co
lumns.}
```

Calculate the number of **false positives** made by the model.

```
In [28]:
```

```
graphlab.canvas.set_target('ipynb')
model_5.show(view='Evaluation')
```

```
In [29]:
```

```
predictions = model_5.predict(validation_data)
len(predictions)
```

```
Out[29]:

9284
```

```
In [30]:
```

```
confusion_matrix = model_5.evaluate(validation_data)['confusion_matrix']
confusion_matrix
```

```
Out[30]:
```

target_label	predicted_label	count
-1	1	1618
-1	-1	3054
1	-1	1463
1	1	3149

[4 rows x 3 columns]

Quiz Question: What is the number of **false positives** on the **validation_data**?

```
In [31]:
```

```
confusion_matrix[(confusion_matrix['target_label'] == -1) & (confusion_matrix['predicted_la
```

```
Out[31]:
```

target_label	predicted_label	count
-1	1	1618

[? rows x 3 columns]
Note: Only the head of the SFrame is printed. This SFrame is lazily evaluated.
You can use sf.materialize() to force materialization.

```
In [32]:
```

```
false_positives = confusion_matrix[(confusion_matrix['target_label'] == -1) & (confusion_ma
print false_positives
```

```
1618
```

Calculate the number of **false negatives** made by the model.

In [33]:

```
false_negatives = confusion_matrix[(confusion_matrix['target_label'] == 1) & (confusion_mat
print false_negatives
```

```
1463
```

In [34]:

```
confusion_matrix[(confusion_matrix['target_label'] == 1) & (confusion_matrix['predicted_lab
```

Out[34]:

target_label	predicted_label	count
1	-1	1463

[? rows x 3 columns]
Note: Only the head of the SFrame is printed. This SFrame is lazily evaluated.
You can use sf.materialize() to force materialization.

Comparison with decision trees

In the earlier assignment, we saw that the prediction accuracy of the decision trees was around **0.64** (rounded). In this assignment, we saw that **model_5** has an accuracy of **0.67** (rounded).

Here, we quantify the benefit of the extra 3% increase in accuracy of **model_5** in comparison with a single decision tree from the original decision tree assignment.

As we explored in the earlier assignment, we calculated the cost of the mistakes made by the model. We again consider the same costs as follows:

- **False negatives**: Assume a cost of $10,000 per false negative.
- **False positives**: Assume a cost of $20,000 per false positive.

Assume that the number of false positives and false negatives for the learned decision tree was

- **False negatives**: 1936
- **False positives**: 1503

Using the costs defined above and the number of false positives and false negatives for the decision tree, we can calculate the total cost of the mistakes made by the decision tree model as follows:

```
cost = $10,000 * 1936  + $20,000 * 1503 = $49,420,000
```

The total cost of the mistakes of the model is $49.42M. That is a **lot of money**!.

Quiz Question: Using the same costs of the false positives and false negatives, what is the cost of the mistakes made by the boosted tree model (**model_5**) as evaluated on the **validation_set**?

In [42]:

```
cost_of_mistakes = (false_negatives * 10000) + (false_positives * 20000)
print cost_of_mistakes
print ("cost = ${:,.2f}".format(cost_of_mistakes))
```

```
46990000
cost = $46,990,000.00
```

Reminder: Compare the cost of the mistakes made by the boosted trees model with the decision tree model. The extra 3% improvement in prediction accuracy can translate to several million dollars! And, it was so easy to get by simply boosting our decision trees.

Most positive & negative loans.

In this section, we will find the loans that are most likely to be predicted **safe**. We can do this in a few steps:

- **Step 1**: Use the **model_5** (the model with 5 trees) and make **probability predictions** for all the loans in the **validation_data**.
- **Step 2**: Similar to what we did in the very first assignment, add the probability predictions as a column called **predictions** into the validation_data.
- **Step 3**: Sort the data (in descreasing order) by the probability predictions.

Start here with **Step 1** & **Step 2**. Make predictions using **model_5** for examples in the **validation_data**. Use `output_type = probability`.

In [44]:

```
validation_data['predictions'] = model_5.predict(validation_data, output_type='probability'
```

Checkpoint: For each row, the probabilities should be a number in the range **[0, 1]**. We have provided a simple check here to make sure your answers are correct.

In [45]:

```
print "Your loans      : %s\n" % validation_data['predictions'].head(4)
print "Expected answer : %s" % [0.4492515948736132, 0.6119100103640573,
                                0.3835981314851436, 0.3693306705994325]
```

```
Your loans      : [0.44925159215927124, 0.6119099855422974, 0.38359811902046
204, 0.3693307042121887]

Expected answer : [0.4492515948736132, 0.6119100103640573, 0.383598131485143
6, 0.3693306705994325]
```

Now, we are ready to go to **Step 3**. You can now use the `prediction` column to sort the loans in **validation_data** (in descending order) by prediction probability. Find the top 5 loans with the highest probability of being predicted as a **safe loan**.

In [46]:

```
validation_data[['grade','predictions']].sort('predictions', ascending = False)[0:5]
```

Out[46]:

grade	predictions
A	0.848508358002
A	0.848508358002
A	0.841295421124
A	0.841295421124
A	0.841295421124

[5 rows x 2 columns]

Quiz Question: What grades are the top 5 loans?

Let us repeat this excercise to find the top 5 loans (in the **validation_data**) with the **lowest probability** of being predicted as a **safe loan**:

In [47]:

```
validation_data[['grade','predictions']].sort('predictions', ascending = True)[0:5]
```

Out[47]:

grade	predictions
D	0.134275108576
C	0.134275108576
B	0.134275108576
C	0.134275108576
C	0.134275108576

[5 rows x 2 columns]

Checkpoint: You should expect to see 5 loans with the grade ['**D**', '**C**', '**C**', '**C**', '**B**'] or with ['**D**', '**C**', '**B**', '**C**', '**C**'].

Effect of adding more trees

In this assignment, we will train 5 different ensemble classifiers in the form of gradient boosted trees. We will train models with 10, 50, 100, 200, and 500 trees. We use the **max_iterations** parameter in the boosted tree module.

Let's get sarted with a model with **max_iterations = 10**:

In [48]:

```
model_10 = graphlab.boosted_trees_classifier.create(train_data, validation_set=None,
        target = target, features = features, max_iterations = 10, verbose=False)
```

Now, train 4 models with **max_iterations** to be:

- `max_iterations = 50`,
- `max_iterations = 100`
- `max_iterations = 200`
- `max_iterations = 500`.

Let us call these models **model_50**, **model_100**, **model_200**, and **model_500**. You can pass in `verbose=False` in order to suppress the printed output.

Warning: This could take a couple of minutes to run.

In [49]:

```
model_50 = graphlab.boosted_trees_classifier.create(train_data, validation_set=None,
        target = target, features = features, max_iterations = 50, verbose=False)
model_100 = graphlab.boosted_trees_classifier.create(train_data, validation_set=None,
        target = target, features = features, max_iterations = 100, verbose=False)
model_200 = graphlab.boosted_trees_classifier.create(train_data, validation_set=None,
        target = target, features = features, max_iterations = 200, verbose=False)
model_500 = graphlab.boosted_trees_classifier.create(train_data, validation_set=None,
        target = target, features = features, max_iterations = 500, verbose=False)
```

Compare accuracy on entire validation set

Now we will compare the predicitve accuracy of our models on the validation set. Evaluate the **accuracy** of the 10, 50, 100, 200, and 500 tree models on the **validation_data**. Use the `.evaluate` method.

In [50]:

```
print "start block"
model_accuracy = []
treeSize = [5, 10, 50, 100, 200, 500]#was just for comparison/experiment purposes.
model_accuracy.append(model_5.evaluate(validation_data)['accuracy'])
print "model_5.evaluate(validation_data)['accuracy'] = ", model_accuracy[0]
model_accuracy.append(model_10.evaluate(validation_data)['accuracy'])
print "model_10.evaluate(validation_data)['accuracy'] = ", model_accuracy[1]
model_accuracy.append(model_50.evaluate(validation_data)['accuracy'])
print "model_50.evaluate(validation_data)['accuracy'] = ", model_accuracy[2]
model_accuracy.append(model_100.evaluate(validation_data)['accuracy'])
print "model_100.evaluate(validation_data)['accuracy'] = ", model_accuracy[3]
model_accuracy.append(model_200.evaluate(validation_data)['accuracy'])
print "model_200.evaluate(validation_data)['accuracy'] = ", model_accuracy[4]
model_accuracy.append(model_500.evaluate(validation_data)['accuracy'])
print "model_500.evaluate(validation_data)['accuracy'] = ", model_accuracy[5]
print "--------------------------------------------------"
minIndex = model_accuracy.index(min(model_accuracy))
print "minimum accuracy = ", min(model_accuracy), " at index = ", minIndex, ", tree size =
maxIndex = model_accuracy.index(max(model_accuracy))
print "maximum accuracy = ", max(model_accuracy), " at index = ", maxIndex, ", tree size =
```

```
start block
model_5.evaluate(validation_data)['accuracy'] =  0.668138733305
model_10.evaluate(validation_data)['accuracy'] =  0.672770357604
model_50.evaluate(validation_data)['accuracy'] =  0.690758293839
model_100.evaluate(validation_data)['accuracy'] =  0.691727703576
model_200.evaluate(validation_data)['accuracy'] =  0.684510986644
model_500.evaluate(validation_data)['accuracy'] =  0.671800947867
--------------------------------------------------
minimum accuracy =  0.668138733305  at index =  0 , tree size =  5
maximum accuracy =  0.691727703576  at index =  3 , tree size =  100
```

Quiz Question: Which model has the **best** accuracy on the **validation_data**?

Quiz Question: Is it always true that the model with the most trees will perform best on test data?

Plot the training and validation error vs. number of trees

Recall from the lecture that the classification error is defined as

$$\text{classification error} = 1 - \text{accuracy}$$

In this section, we will plot the **training and validation errors versus the number of trees** to get a sense of how these models are performing. We will compare the 10, 50, 100, 200, and 500 tree models. You will need matplotlib (http://matplotlib.org/downloads.html) in order to visualize the plots.

First, make sure this block of code runs on your computer.

In [51]:

```
import matplotlib.pyplot as plt
%matplotlib inline
def make_figure(dim, title, xlabel, ylabel, legend):
    plt.rcParams['figure.figsize'] = dim
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    if legend is not None:
        plt.legend(loc=legend, prop={'size':15})
    plt.rcParams.update({'font.size': 16})
    plt.tight_layout()
```

In order to plot the classification errors (on the **train_data** and **validation_data**) versus the number of trees, we will need lists of these accuracies, which we get by applying the method `.evaluate`.

Steps to follow:

- **Step 1:** Calculate the classification error for model on the training data (**train_data**).
- **Step 2:** Store the training errors into a list (called `training_errors`) that looks like this:

  ```
  [train_err_10, train_err_50, ..., train_err_500]
  ```

- **Step 3:** Calculate the classification error of each model on the validation data (**validation_data**).
- **Step 4:** Store the validation classification error into a list (called `validation_errors`) that looks like this:

  ```
  [validation_err_10, validation_err_50, ..., validation_err_500]
  ```

  Once that has been completed, the rest of the code should be able to evaluate correctly and generate the plot.

Let us start with **Step 1**. Write code to compute the classification error on the **train_data** for models **model_10**, **model_50**, **model_100**, **model_200**, and **model_500**.

In [52]:

```
train_err_10 = 1 - model_10.evaluate(train_data)['accuracy']
train_err_50 = 1 - model_50.evaluate(train_data)['accuracy']
train_err_100 = 1 - model_100.evaluate(train_data)['accuracy']
train_err_200 = 1 - model_200.evaluate(train_data)['accuracy']
train_err_500 = 1 - model_500.evaluate(train_data)['accuracy']
```

Now, let us run **Step 2**. Save the training errors into a list called **training_errors**

In [53]:

```
training_errors = [train_err_10, train_err_50, train_err_100,
                   train_err_200, train_err_500]
```

Now, onto **Step 3**. Write code to compute the classification error on the **validation_data** for models **model_10**, **model_50**, **model_100**, **model_200**, and **model_500**.

```
In [54]:
```

```
validation_err_10 = 1 - model_10.evaluate(validation_data)['accuracy']
validation_err_50 = 1 - model_50.evaluate(validation_data)['accuracy']
validation_err_100 = 1 - model_100.evaluate(validation_data)['accuracy']
validation_err_200 = 1 - model_200.evaluate(validation_data)['accuracy']
validation_err_500 = 1 - model_500.evaluate(validation_data)['accuracy']
```

Now, let us run **Step 4**. Save the training errors into a list called **validation_errors**

```
In [55]:
```

```
validation_errors = [validation_err_10, validation_err_50, validation_err_100,
                     validation_err_200, validation_err_500]
```

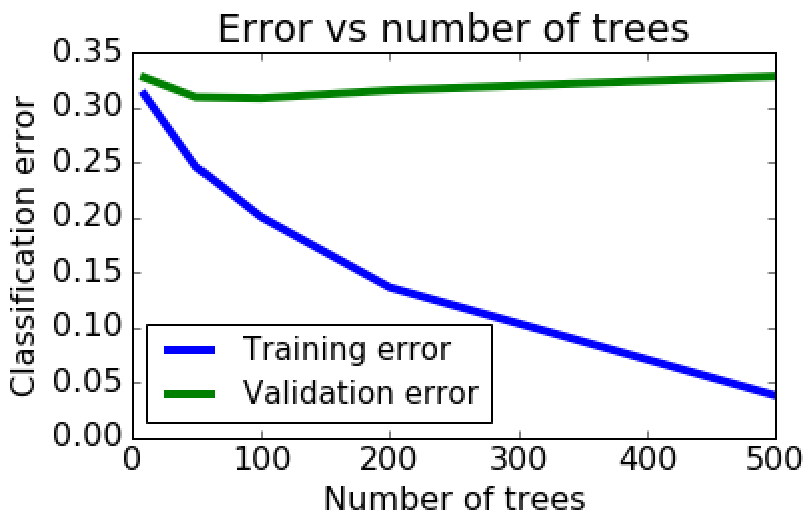
Now, we will plot the **training_errors** and **validation_errors** versus the number of trees. We will compare the 10, 50, 100, 200, and 500 tree models. We provide some plotting code to visualize the plots within this notebook.

Run the following code to visualize the plots.

```
In [56]:
```

```
plt.plot([10, 50, 100, 200, 500], training_errors, linewidth=4.0, label='Training error')
plt.plot([10, 50, 100, 200, 500], validation_errors, linewidth=4.0, label='Validation error

make_figure(dim=(10,5), title='Error vs number of trees',
            xlabel='Number of trees',
            ylabel='Classification error',
            legend='best')
```



Quiz Question: Does the training error reduce as the number of trees increases?

Quiz Question: Is it always true that the validation error will reduce as the number of trees increases?

Quiz

1. What percentage of the predictions on sample_validation_data did model_5 get correct?

- 25%
- 50%
- [✓ 75%]
- 100%

2. According to **model_5**, which loan is the least likely to be a safe loan?

- First
- Second
- [✓ Third]
- Fourth

3. What is the number of false positives on the validation data?

1618

4. Using the same costs of the false positives and false negatives, what is the cost of the mistakes made by the boosted tree model (model_5) as evaluated on the validation_set?

46990000

5. What grades are the top 5 loans?

- [✓ A]
- B
- C
- D
- E

6. Which model has the best accuracy on the validation_data?

- model_10
- model_50
- [✓ model_100]
- model_200
- model_500

7. Is it always true that the model with the most trees will perform best on the test/validation set?

- Yes, a model with more trees will ALWAYS perform better on the test/validation set.
- [✓ No, a model with more trees does not always perform better on the test/validation set.]

8. Does the training error reduce as the number of trees increases?

- [✓ Yes]
- No

9. Is it always true that the test/validation error will reduce as the number of trees increases?

- Yes, it is ALWAYS true that the test/validation error will reduce as the number of trees increases.
- [✓ No, the test/validation error will not necessarily always reduce as the number of trees increases.]