

How do I build a prediction model for Lending Club loan defaults?

Introduction (5 mts)

Business Context. We have seen in the previous case what we can do to determine if one aspect of personal information (e.g. whether the income information is verified) is important when predicting default status. This type of analysis certainly helps with simplifying the lending process. However, the biggest question for every P2P lending company — whether a user will default or not — is still not answered. The algorithm to predict default probability is the core of our business and we need to build it carefully.

Business Problem. Your task is to build a classification model for determining whether a user will default on their loan or not.

Analytical Context. As a data scientist, you now need to focus on honing the classification algorithm. You need to verify quantitatively that the algorithm is going to work for future data. In addition, you need to expand your toolkit to not rely solely on logistic regression, as it has some shortcomings which are better handled by other types of models. The final classification algorithm will be selected via the model evaluation procedures we introduced in the previous cases. We will consider models generated from logistic regression, as well as two new algorithms: Naive Bayes classifiers and random forests.

The case is structured as follows: you will: 1) generate a baseline model from logistic regression; 2) introduce Naive Bayes classifiers and random forests; 3) use these algorithms to upgrade our model; and finally 4) use cross-validation to select the best model.

Evaluating our logistic model (20 mts)

In [32]: ! pip install sympy

Requirement already satisfied: sympy in /root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages (1.6)
Requirement already satisfied: mpmath>=0.19 in /root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages (from sympy) (1.1.0)
WARNING: You are using pip version 20.0.2; however, version 20.1.1 is available.
You should consider upgrading via the '/root/.virtualenvs/ds4a-py3/bin/python -m pip install --upgrade pip' command.

```
In [33]: import pandas as pd
import numpy as np
from scipy.stats import chi2_contingency, norm
import statsmodels.api as sm
from sklearn.tree import export_graphviz
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, auc, accuracy_score
from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.externals.six import StringIO
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
from matplotlib.patches import Patch
from scipy import interp
from statsmodels.formula.api import ols
import seaborn as sns

import pydotplus
from IPython.display import SVG
from graphviz import Source
from IPython.display import Image
from sympy import var, plot_implicit, Eq

#ignore log(0) and divide by 0 warning
np.seterr(divide = 'ignore');
```

```
In [34]: df = pd.read_csv("Lending_club_cleaned_2.csv", \
                        dtype = {"loan_status":"category", "annual_inc":"float", \
                                "verification_status":"category", "emp_length":"category", \
                                "home_ownership":"category", "int_rate":"object", \
                                "loan_amnt":"float", "purpose":"category", \
                                "term":"category", "grade":"category"})
df.int_rate = df.int_rate.str.rstrip("%").astype("float")
```

```
In [35]: df.head()
```

Out[35]:

	loan_status	annual_inc	verification_status	emp_length	home_ownership	int_rate	loan_amnt
0	Fully Paid	24000.0	Verified	10+ years	RENT	10.65	5000.0
1	Charged Off	30000.0	Source Verified	< 1 year	RENT	15.27	2500.0
2	Fully Paid	12252.0	Not Verified	10+ years	RENT	15.96	2400.0
3	Fully Paid	49200.0	Source Verified	10+ years	RENT	13.49	10000.0
4	Fully Paid	80000.0	Source Verified	1 year	RENT	12.69	3000.0

Before we proceed, let's review the results and code for the final model we used in the previous case.

```
In [36]: # design matrix for the data
df_log2 = pd.concat([(df.loan_amnt - df.loan_amnt.mean())/df.loan_amnt.
std(), \
                    (df.int_rate - df.int_rate.mean())/df.int_rate.std
()), \
```

```

nc.std(),\
                                (df.annual_inc - df.annual_inc.mean())/df.annual_i
pd.get_dummies(df.home_ownership, prefix="home", d
rop_first=True), \
pd.get_dummies(df.purpose, prefix="purpose", drop_
first=True), \
pd.get_dummies(df.grade, prefix="grade",drop_first
=True)], axis=1)
df_log2["verification_status"] = df.verification_status.cat.codes
df_log2["emp_length"] = df.emp_length.cat.codes
df_log2["term"] = df.term.cat.codes
df_log2["Intercept"] = 1

# fit the logistic regression model
logit_full1 = sm.Logit(df.loan_status.cat.codes, df_log2)
logit_full1_res = logit_full1.fit()

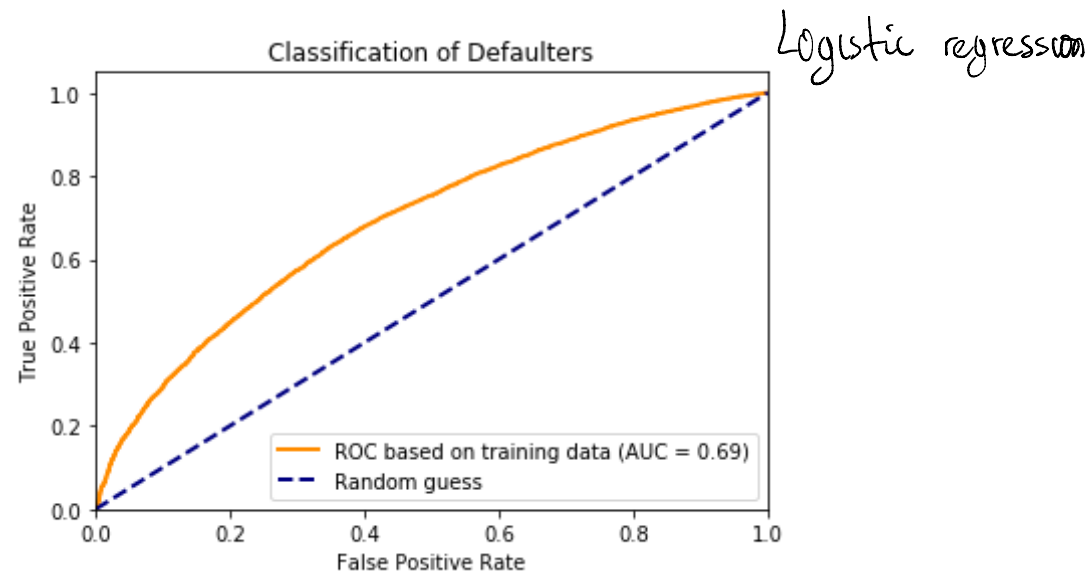
# plot roc
predict_withvs = logit_full1_res.predict(df_log2)

roc_vs = roc_curve(df.loan_status.cat.codes, predict_withvs)
auc_vs = auc( roc_vs[0], roc_vs[1] )

plt.figure()
lw = 2
plt.plot(roc_vs[0], roc_vs[1], color='darkorange',
         lw=lw, label='ROC based on training data (AUC = %0.2f)' % auc_
vs)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--', label='Ra
ndom guess')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Classification of Defaulter')
plt.legend(loc="lower right")
plt.show()

```

Optimization terminated successfully.
Current function value: 0.378195



Exercise 1: (15 mts)

Generate the 10-fold cross-validation ROC curve (the average of all 10 ROC curves) and calculate the average AUC. Plot the cross-validation ROC curve, the training data ROC curve, and the random guess line on a single graph.

Answer. One possible solution is shown below:

```
In [37]: # generate cross-validated ROC and AUC for logistic regression model

def cv_roc(X, y, n_split):
```

```

cv = StratifiedKFold(n_split)

tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)
for train, test in cv.split(X, y):
    logit = sm.Logit(y.iloc[train], X.iloc[train])
    logit_res = logit.fit(dis=0)
    probas_ = logit_res.predict(X.iloc[test])
    # Compute ROC curve and area under the curve
    fpr, tpr, thresholds = roc_curve(y.iloc[test], probas_)
    tprs.append(interp(mean_fpr, fpr, tpr))
    tprs[-1][0] = 0.0
    roc_auc = auc(fpr, tpr)
    aucs.append(roc_auc)

return tprs, aucs, mean_fpr

```

```

In [38]: #plot the cross-validated ROC
tprs_vs, aucs_vs, mean_fpr_vs = cv_roc(df_log2, df_loan_status.cat.codes, 10)

plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='blue',
         label='Random guess', alpha=.8)

mean_tpr_vs = np.mean(tprs_vs, axis=0)
mean_tpr_vs[-1] = 1.0

mean_auc_vs = auc(mean_fpr_vs, mean_tpr_vs)
std_auc_vs = np.std(aucs_vs)

plt.plot(mean_fpr_vs, mean_tpr_vs, color='darkorange',
         label=r'Cross-validated Mean ROC (AUC = %0.2f $\pm$ %0.2f)' %
         (mean_auc_vs, std_auc_vs),
         lw=2, alpha=.8)

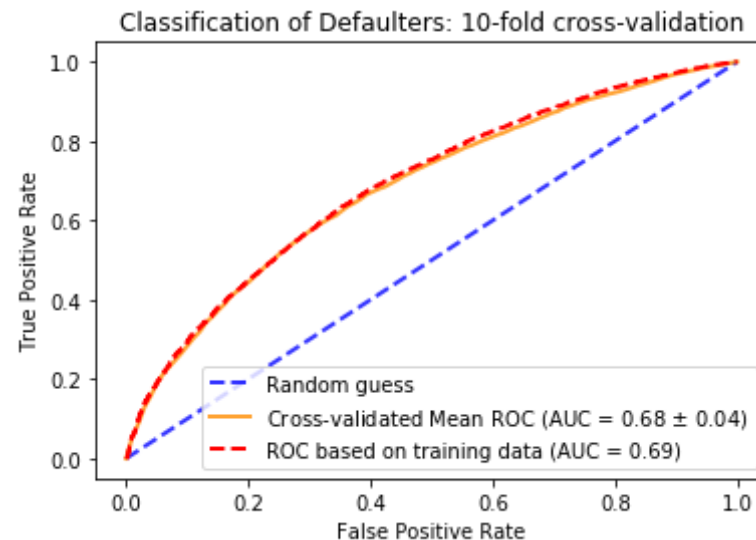
plt.plot(roc_vs[0], roc_vs[1], color='red', linestyle='--',
         lw=lw, label='ROC based on training data (AUC = %0.2f)' % auc_

```

vs)

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.title('Classification of Defaulters: 10-fold cross-validation')
plt.show()
```

```
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages/ipykernel_launcher.py:16: DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0, use numpy.interp instead
app.launch_new_instance()
```



We can see the average AUC is a bit smaller than the AUC calculated with training data. The difference is small, indicating the overfitting problem is not material in our case. The cross-validation ROC curve is also very close to the ROC curve generated with training data, further strengthening our contention that we do not have an overfitting problem.

Although logistic regression is one of the most commonly used classification algorithms, it is not the only one. If the underlying relationship between the logit of odds and the covariates is not

linear, we need to consider higher order terms of the covariates to make the logistic model valid. Doing so requires a lot of tuning and experimenting. In such cases, it is often better to look at a classification algorithm that does not rely on a specific assumption about the relationship between the probability of the outcome and the covariates. One such model is the **Naive Bayes classifier**.

The Naive Bayes classifier (65 mts)

Conditional probability and Bayes theorem (5 mts)

Naive Bayes depends on the concept of **conditional probability (CP)**. The conditional probability of event A given event B is defined as the probability of observing event A provided that event B has already happened. Numerically, this means:

$$\text{CP}(A \text{ given } B) = \frac{P(A \text{ and } B)}{P(B)}.$$

Of course, there is nothing special about A or B , so we can reverse them in the identity above to get:

$$\text{CP}(B \text{ given } A) = \frac{P(B \text{ and } A)}{P(A)}.$$

Notice that the numerators on the right-hand side of both equations are identical. Therefore, we can equate them and substitute in the remainder of each identity to get:

$$\begin{aligned} \text{CP}(A \text{ given } B)P(B) &= \text{CP}(B \text{ given } A)P(A) \\ \rightarrow \text{CP}(A \text{ given } B) &= \frac{\text{CP}(B \text{ given } A)P(A)}{P(B)} \end{aligned}$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

This is the famous **Bayes theorem**, upon which the Naive Bayes classifier is constructed.

Exercise 2: (5 mts)

Assume we know that the probability that a customer had no verified income source is 0.4, the probability of default is 0.5 and the conditional probability of default given the customer had no verified income source is 0.8. What is the probability of having no verified income given the customer is a defaulter?

Answer. Using the Bayes theorem, we have

$$\text{CP}(\text{no verified income given defaulter}) = \frac{\text{CP}(\text{defaulter given no verified income}) \times \text{P}(\text{no verified income})}{\text{P}(\text{defaulter})} = \frac{0.8 * 0.4}{0.5}$$

Introducing the algorithm (10 mts)

Recall that in logistic regression, the main quantity of interest is the probability of default given specific values for the covariates. We explicitly modeled this probability as a linear function of covariates after transformation by the logit function.

The Naive Bayes classifier aims to estimate this probability from a different perspective. From the Bayes theorem, we can express this conditional probability in the following way:

$$\text{CP}(\text{defaulter given covariates}=\text{values}) = \frac{\text{CP}(\text{covariates}=\text{values given defaulter})}{\text{P}(\text{covariates}=\text{values})}$$

Therefore, if we can just estimate the values on the right-hand side, we can get what we want. Let's take a look at each of these quantities:

1. $\text{P}(\text{defaulter})$. This is the probability of having a defaulter. It can be easily estimated as the proportion of defaulters in the dataset.
2. $\text{P}(\text{covariates}=\text{values})$. This is the probability of having a customer whose covariates are equal to `values`. This can also be estimated by proportions, but we will actually find out that it is not necessary to estimate this quantity at all (more on this later).
3. $\text{CP}(\text{covariates}=\text{values given defaulter})$. We can estimate this by looking at the proportion of defaulters with covariates equal to `values`. But if we have a lot of covariates,

the number of such defaulters might be extremely small or even equal to zero. The main challenge in building a Naive Bayes classifier is estimating this particular probability.

Before we proceed to talk about the algorithm to estimate $CP(\text{covariates}=\text{values given defaulter})$, let's first understand why, when the number of covariates is large, it is hard to estimate $CP(\text{covariates}=\text{values given defaulter})$ by proportions directly.

Exercise 3 (5 mts)

Assume we have 10 covariates and that all of them are binary variables. What is the minimum number of samples we need for it to be possible to have at least one sample for each possible combination of values of these covariates?

Answer. We only need to calculate how many different possible values of covariates we can have when 10 binary covariates are present. The answer is $2^{10} = 1024$ because each covariate has two different values and we have 10 covariates in total. So we need to have at least 1024 samples for it to even be possible to estimate $CP(\text{covariates}=\text{values given defaulter})$.

This is a very large value for only 10 variables with very few possible values. If we suddenly had 20 variables, this number goes up to 10^6 and becomes impossible to reach.

Why is the classifier "naive"? (5 mts)

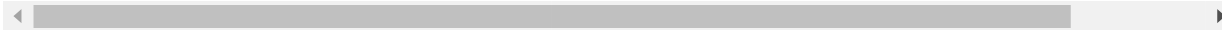
As we have seen in Exercise 3, we need a lot of samples to be able to estimate $CP(\text{covariates}=\text{values given defaulter})$ for all possible values using observed proportions. In order to get around this problem, the Naive Bayes classifier makes the assumption that *all covariates are independent when conditioned on the outcome*. This assumption might never be true in the real world, which is why the method is called "naive".

Nonetheless, it usually does not severely hamper the tool in practice and gives quite powerful results a lot of the time.

In our case, the assumption implies that within defaulters/non-defaulters, all covariates are independently distributed:

$$\begin{aligned} \text{CP}(\text{covariates}=\text{values given defaulter}) &= \text{CP}(\text{covariate}_1=\text{value}_1 \text{ given defaulter}) \\ &\times \text{CP}(\text{covariate}_p=\text{value}_p \text{ given defaulter}) \end{aligned}$$

Here we assume there are p covariates in total. Let's see why it reduces the number of samples required to estimate proportions.



Exercise 4: (5 mts)

Under the same assumptions as in Exercise 3, plus the independence assumption, how many samples do we need to have in order to estimate the conditional probability $\text{CP}(\text{covariates}=\text{values given defaulter})$?

Answer. We need to estimate p simpler conditional probabilities and each one of them only concerns one binary variable, which has 2 possible outcomes. So we only need at least $2 \times 10 = 20$ samples now, which is *much* smaller than 1024.

With this assumption, we estimate $\text{CP}(\text{covariate}_1=\text{value}_1 \text{ given defaulter})$ to $\text{CP}(\text{covariate}_p=\text{value}_p \text{ given defaulter})$ instead of $\text{CP}(\text{covariates}=\text{values given defaulter})$. When the covariate under consideration is discrete, we just need to estimate the proportions of its unique values in defaulters to get $\text{CP}(\text{covariate}=\text{value given defaulter})$.

Exercise 5: (5 mts)

Using the available data, estimate $\text{CP}(\text{term given defaulter})$ and $\text{CP}(\text{verification_status given defaulter})$ for all different values of `term` and

```
verification_status.
```

Answer. Shown below:

```
In [17]: # conditional probabilities of term and verification_status
df_default = df[df.loan_status=="Charged Off"]
# term
print("Conditional probability of term given defaulter:")
print((df_default.term.value_counts()/df_default.shape[0]));
print("")
# verification_status
print("Conditional probability of verification_status given defaulter:")
print(df_default.verification_status.value_counts()/df_default.shape[0]);
```

Conditional probability of term given defaulter:

36 months	0.56489
60 months	0.43511

Name: term, dtype: float64

Conditional probability of verification_status given defaulter:

Not Verified	0.376838
Verified	0.363419
Source Verified	0.259743

Name: verification_status, dtype: float64

When a covariate is continuous, the "proportion" of a given value is usually zero due to the continuous nature of the covariate. To get around this, we use the density of a covariate value to represent the conditional probability. The density depends on the distribution of the overall data; **in the Naive Bayes classifier, we usually assume that the data is normally distribution** (the second reason why it's naive). We then find the parameters of a normal distribution that is most closely aligned with the observed distribution of the covariate to get $CP(\text{covariate}=\text{value} \text{ given defaulter})$.

We know that a normal distribution is uniquely determined by its mean and variance, so the normal distribution with mean equal to the sample mean and variance equal to the sample variance of the covariate is a good estimate to use. Once we get this normal distribution, the density can be calculated directly for any covariate value.

Exercise 6: (5 mts)

Find the normal distribution that most aligned with the observed distribution of `annual_inc` in defaulters. Based on this normal distribution, calculate $CP(\text{annual_inc}=10,000 \text{ given defaulter})$. Function `pdf()` in `scipy.stats.norm` calculates the density of a normal distribution at a given value.

Answer. Shown below:

```
In [18]: mean_est = df_default.annual_inc.mean()
var_est = df_default.annual_inc.var()

# pdf takes standard deviation as scale parameter
# note we have log10 transformed annual_inc so when call norm.pdf, we should evaluate at 4
print("CP(annual_inc=10,000 given defaulter) =", norm.pdf(10000, loc = mean_est, scale = np.sqrt(var_est)))
```

```
CP(annual_inc=10,000 given defaulter) = 4.473123137400818e-06
```

Once we have estimated all these single covariate conditional probabilities, we multiply them together to get our final estimate of $CP(\text{covariates}=\text{values given defaulter})$. Now, we claim that we can use the following procedure to get $CP(\text{defaulter given covariates}=\text{values})$:

1. Estimate $P(\text{defaulter})$ and $P(\text{non-defaulter})$.
2. Estimate $CP(\text{covariates}=\text{value given defaulter})$ and $CP(\text{covariates}=\text{value given non-defaulter})$
3. Estimate $CP(\text{defaulter given covariates}=\text{values})$ with the following equation:

$$= \frac{CP(\text{defaulter given covariates}=\text{values})}{CP(\text{covariates}=\text{values given defaulter})P(\text{defaulter}) + CP(\text{covariates}=\text{values given non-defaulter})P(\text{non-defaulter})}$$

Exercise 7: (10 mts)

Provide a justification for the equation in step 3.

Answer. Since one can only be a defaulter or a non-defaulter, we have

$$CP(\text{defaulter given covariates}=\text{values}) + CP(\text{non-defaulter given covariates}=\text{values}) = 1$$

With Bayes theorem, we have

$$\begin{aligned} CP(\text{defaulter given covariates}=\text{values}) &= \frac{CP(\text{covariates}=\text{values given defaulter})P(\text{defaulter})}{P(\text{covariates}=\text{values})} \\ &= \frac{CP(\text{non-defaulter given covariates}=\text{values})}{\frac{CP(\text{covariates}=\text{values given non-defaulter})P(\text{non-defaulter})}{P(\text{covariates}=\text{values})}} \end{aligned}$$

Taking the ratio of the above two equations, we have

$$\begin{aligned} &\frac{CP(\text{defaulter given covariates}=\text{values})}{CP(\text{non-defaulter given covariates}=\text{values})} \\ &= \frac{CP(\text{covariates}=\text{values given defaulter})P(\text{defaulter})}{CP(\text{covariates}=\text{values given non-defaulter})P(\text{non-defaulter})} \end{aligned}$$

Combining the above equation with

$$CP(\text{defaulter given covariates}=\text{values}) + CP(\text{non-defaulter given covariates}=\text{values}) = 1$$

we get the equation in Step 3.

Let's pause a moment here and think about what the Naive Bayes estimator really does. From the step 3 we can see the conditional probability of default is indeed determined by the values of covariates through the conditional probabilities $CP(\text{covariates}=\text{values given defaulter})$ and $CP(\text{covariates}=\text{values given non-defaulter})$. The ratio between these two quantities reflects how different the distribution of covariates is in defaulters vs. non-defaulters. Effectively, Naive Bayes is comparing the distribution of covariates in groups of samples with different outcomes. This provides us a way to pre-screen for variables that should be included in the Naive Bayes classifier.

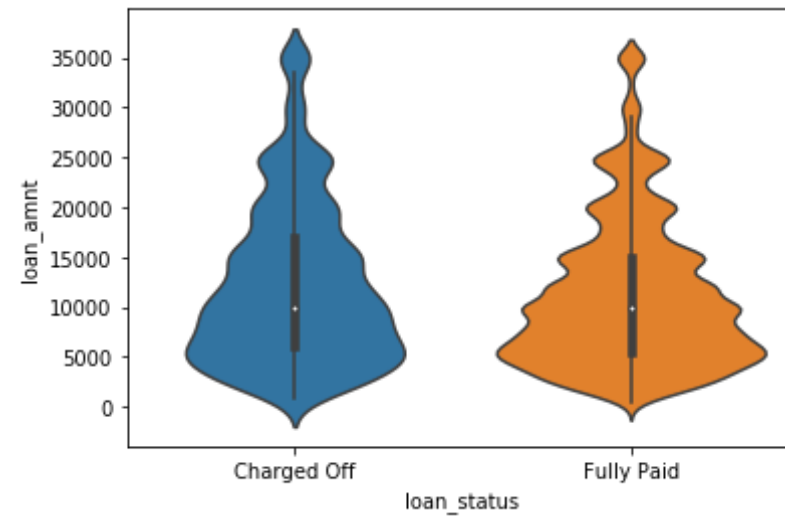
Let's visualize the distributions of covariates in our dataset, stratified by loan status, to get an idea of which should be included.

Exercise 8: (10 mts)

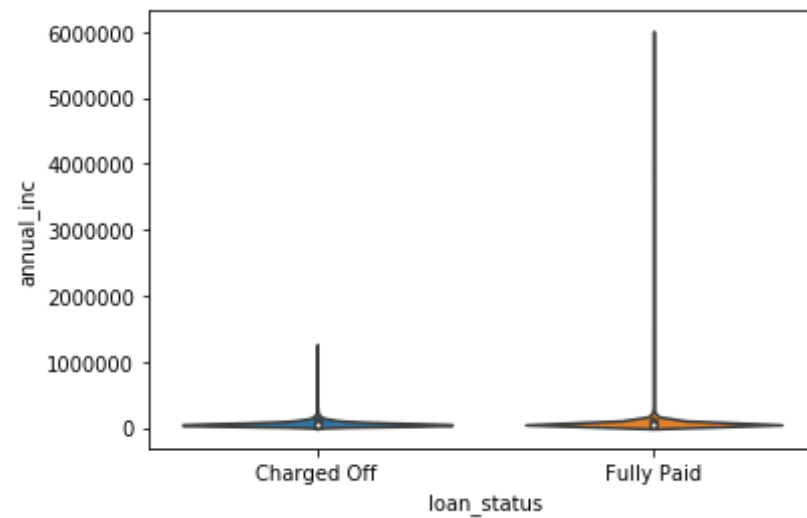
Use the `boxplot()` function in the `seaborn` package to visualize the distribution of the following continuous variables: `loan_amnt`, `annual_inc` and `int_rate` in defaulters and non-defaulters. Do you spot any differences in the distributions in defaulters and non-defaulters? Do these seem to satisfy the assumptions of the Naive Bayes classifier? If not, what should you do to get them to conform?

Answer. One possible solution is shown below:

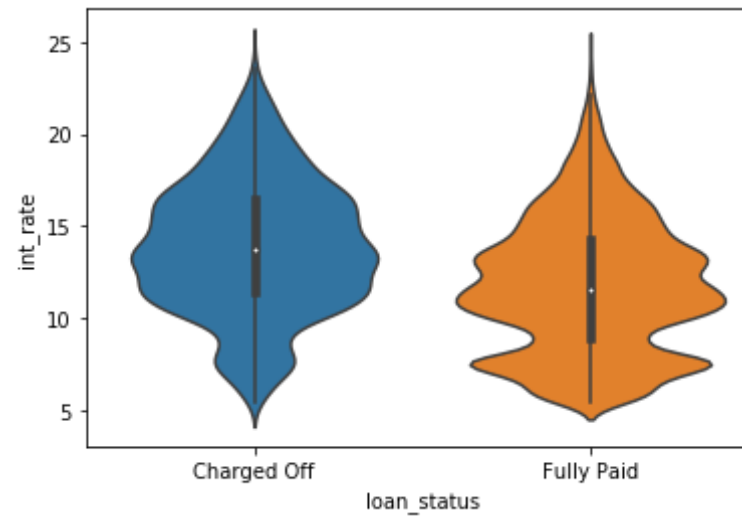
```
In [19]: # plot the distribution of the continuous variable
sns.violinplot(x = "loan_status", y = "loan_amnt", data = df);
```



```
In [20]: sns.violinplot(x = df.loan_status, y = df.annual_inc);
```

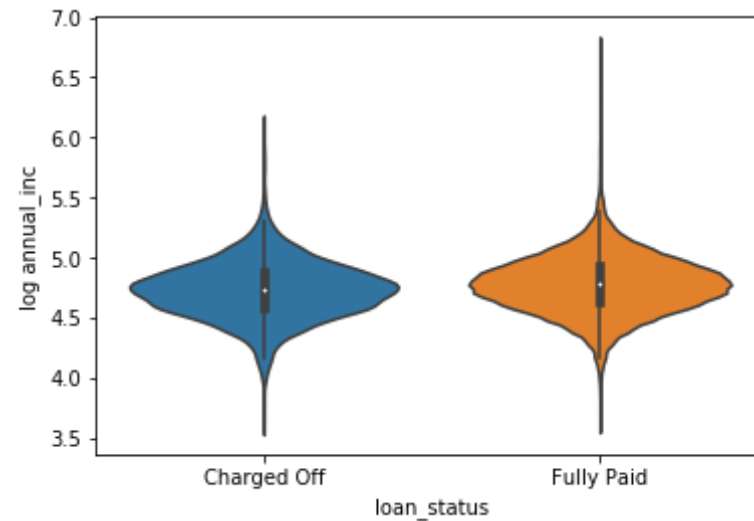


```
In [21]: sns.violinplot(x = "loan_status", y = "int_rate", data = df );
```

We can see that the distribution of interest rate is different in the two types of users. The distribution of loan amount is also different, where non-defaulters tend to have more loans with small amounts (around 5000). The distributions of annual income seem to be heavily skewed, so let's apply a logarithmic transformation to see if we can get these to look more normal:

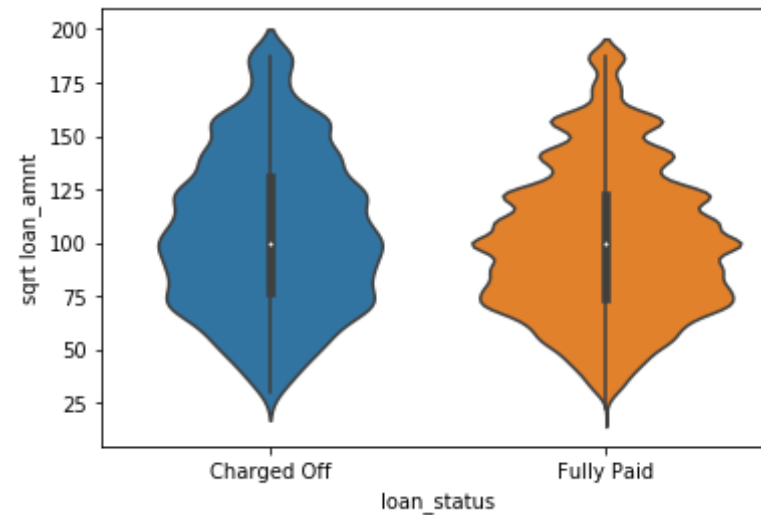
```
In [22]: ax = sns.violinplot(x = df.loan_status, y = np.log10(df.annual_inc));  
ax.set(ylabel='log annual_inc');
```



This is much better. The difference in annual income is not very large, but we can still see a slight increase in the mean of the annual income in non-defaulters compared to defaulters.

The distribution for `int_rate` is close enough to normal, but that of `loan_amnt` seems a bit skewed (although nowhere near the degree of `annual_inc`). Let's apply a square root transformation:

```
In [23]: ax = sns.violinplot(x = df.loan_status, y = np.sqrt(df.loan_amnt));  
ax.set(ylabel='sqrt loan_amnt');
```



This looks better.

From the above exercise, we find that we need to transform `annual_inc` and `loan_amnt` so that the normality assumption can be approximately satisfied. We will use the transformed variables to construct the Naive Bayes estimator:

```
In [24]: df["annual_inc_log"] = np.log10(df.annual_inc)
df["loan_amnt_sqrt"] = np.sqrt(df.loan_amnt)
```

Implementing the classifier (30 mts)

We have seen that in order to estimate the classifier, we need to estimate the normal distribution that can be fit to each variable, then use this to estimate the probabilities we need. The below function `nb_fit_con()` will accomplish the first task:

```
In [25]: """
Learn parameters for continuous covariate
y is a binary outcome
```

```

X is a pandas series of a continuous covariate
"""
def nb_fit_con(y, X):
    y_code = y.cat.codes
    X_0 = X[y_code==0]
    X_1 = X[y_code==1]
    return [[np.mean(X_0), np.std(X_0)], [np.mean(X_1), np.std(X_1)]]

```

Exercise 9: (5 mts)

Write a function `nb_fit_dis` to estimate the probabilities of each level of a discrete covariate in defaulters and non-defaulters. The function returns a list of two lists. The first element contains probabilities in defaulters. The second element contains probabilities in non-defaulters.

Answer. One possible solution is shown below:

```

In [26]: """
Learn parameters for discrete covariate
y is a binary outcome
X is a pandas series of a discrete covariate
"""
def nb_fit_dis(y, X):
    y_code = y.cat.codes
    X_code = X.cat.codes
    X_nlvl = len(X.cat.categories)

    X_0 = X_code[y_code==0]
    X_1 = X_code[y_code==1]

    X_0_freq = [np.sum(X_0==x)/len(X_0) for x in range(X_nlvl)]
    X_1_freq = [np.sum(X_1==x)/len(X_1) for x in range(X_nlvl)]
    return [X_0_freq, X_1_freq]

```

The following function uses the two estimation functions we wrote above to train a Naive Bayes classifier:

```
In [27]: """
y is a binary outcome
X is a pandas dataframe of covariates
"""
def train_nb_classifier( y, X ):
    trained_model = {}
    for ind_col in X.columns:
        # check data type
        if X[ind_col].dtype=="float":
            params_fit = nb_fit_con(y, X[ind_col])
        else:
            params_fit = nb_fit_dis(y, X[ind_col])
        trained_model[ind_col] = params_fit
    margin_p_0 = np.sum(y.cat.codes==0)/len(y)
    trained_model["margin_p"] = [margin_p_0, 1-margin_p_0]
    return trained_model
```

Once the model is trained, the following function `predict_nb_classifier` predicts the probability of not defaulting for new samples:

```
In [28]: """
log of normal density
"""
def log_dnorm( x, loc, scale ):
    return -(x-loc)**2/2/(scale**2) - np.log(scale) - 0.5*np.log(2*np.p
i)

"""
prediction
X_test: a pandas dataframe of covariates for prediction
discrete vars have the same coding scheme as in the train datas
et
"""

def predict_nb_classifier( nb_fit, X_test ):
    pred_0_p = np.ones( X_test.shape[0] ) * np.log(nb_fit["margin_p"][0])
    pred_1_p = np.ones( X_test.shape[0] ) * np.log(nb_fit["margin_p"][1])
```

```

for ind_col in X_test.columns:
    params = nb_fit[ind_col]
    if X_test[ind_col].dtype=="float":
        #increment prob for outcome 0 for all samples
        update_0 = log_dnorm( X_test[ind_col], loc = params[0][0],
scale = params[0][1] )
        update_1 = log_dnorm( X_test[ind_col], loc = params[1][0],
scale = params[1][1] )
    else:
        update_0 = [np.log(params[0][x]) for x in X_test[ind_col].c
at.codes]
        update_1 = [np.log(params[1][x]) for x in X_test[ind_col].c
at.codes]
        pred_0_p = pred_0_p + update_0
        pred_1_p = pred_1_p + update_1

#calculate the estimated probability of outcome=1
#specifically handle Inf
p_ratio = np.exp(pred_1_p-pred_0_p)
p_1 = p_ratio/(1+p_ratio)
p_1[p_ratio==np.Inf] = 1

return p_1

```

Exercise 10: (10 mts)

Use functions `train_nb_classifier` and `predict_nb_classifier` to write a function to generate ROC curve and calculate AUC for NB classifier with `n_split` -fold cross validation (refer to `cv_roc` for a template). The input should be `X`, the matrix of covariates, `y`, the vector of outcomes and `n_split`, the number of folds. The output is the same 3-tuple as function `cv_roc`, where the first element is the list of true positive rates across all `n_split` folds, the second element is a list of AUCs and the third element is a list of mean false positive rate which is set to be `np.linspace(0, 1, 100)`. Use this function on the NB classifier with covariate `term`, `verification_status` and `annual_inc`. Compare this to the logistic model using the same set of covariates.

```
In [29]: def cv_roc_nb(X, y, n_split):

    cv = StratifiedKFold(n_split)

    tprs = []
    aucs = []
    mean_fpr = np.linspace(0, 1, 100)
    for train, test in cv.split(X, y):
        nb_trained = train_nb_classifier(y.iloc[train], X.iloc[train])
        probas_ = predict_nb_classifier( nb_trained, X.iloc[test] )
        # Compute ROC curve and area the curve
        fpr, tpr, thresholds = roc_curve(y.iloc[test].cat.codes, probas_
_)
        tprs.append(interp(mean_fpr, fpr, tpr))
        tprs[-1][0] = 0.0
        roc_auc = auc(fpr, tpr)
        aucs.append(roc_auc)

    return tprs, aucs, mean_fpr
```

```
In [30]: # fit NB
tprs_nb, aucs_nb, mean_fpr_nb = cv_roc_nb( df[["term","verification_sta
tus","annual_inc_log"]], df.loan_status, 5)
# fit logistic
tprs_lgs, aucs_lgs, mean_fpr_lgs = cv_roc(df_log2[["Intercept","term",
"verification_status","annual_inc"]], df.loan_status.cat.codes, 5)

plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='blue',
        label='Random guess', alpha=.8)

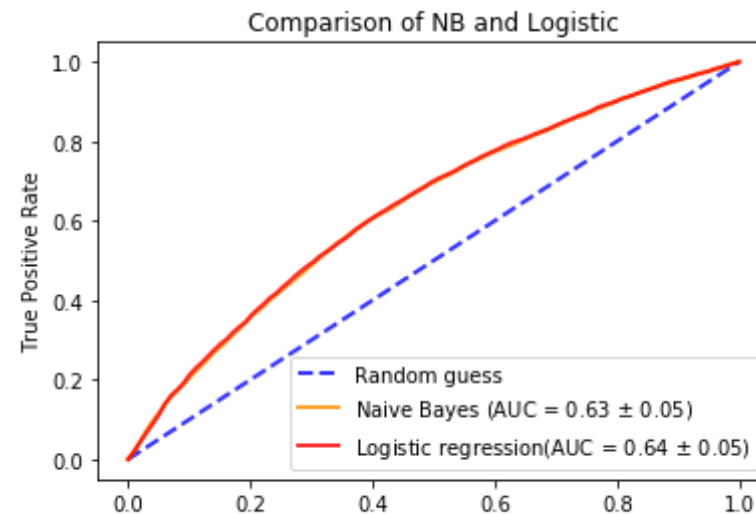
mean_tpr_nb = np.mean(tprs_nb, axis=0)
mean_tpr_nb[-1] = 1.0
mean_tpr_lgs = np.mean(tprs_lgs, axis=0)
mean_tpr_lgs[-1] = 1.0

mean_auc_nb = auc(mean_fpr_nb, mean_tpr_nb)
std_auc_nb = np.std(aucs_nb)
mean_auc_lgs = auc(mean_fpr_lgs, mean_tpr_lgs)
std_auc_lgs = np.std(aucs_lgs)
```

```
plt.plot(mean_fpr_nb, mean_tpr_nb, color='darkorange',
        label=r'Naive Bayes (AUC = %0.2f  $\pm$  %0.2f)' % (mean_auc_nb,
        std_auc_nb),
        lw=2, alpha=.8)
plt.plot(mean_fpr_lgs, mean_tpr_lgs, color='red',
        label=r'Logistic regression(AUC = %0.2f  $\pm$  %0.2f)' % (mean_
        auc_lgs, std_auc_lgs),
        lw=2, alpha=.8)

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.title('Comparison of NB and Logistic')
plt.show()
```

```
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages/ipykernel_launcher.py:13: DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0, use numpy.interp instead
del sys.path[0]
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages/ipykernel_launcher.py:16: DeprecationWarning: scipy.interp is deprecated and will be removed in SciPy 2.0.0, use numpy.interp instead
app.launch_new_instance()
```



From Exercise 10, we can see that the Naive Bayes classifier has a slightly lower AUC than the corresponding logistic model. This might be due to the fact that these three covariates are highly correlated (as we have seen in the previous case). But the difference is not very large. Considering the fact that the Naive Bayes classifier is very easy to implement, we can still use it as a preliminary classifier.

Which one to use? (5 mts)

When deciding which model to use, the gold standard should always include the prediction performance; e.g. AUC. Aside from the performance, there are many other factors one should consider when choosing between these two models:

1. Purpose of the analysis. If a quick and dirty classifier is what we want, Naive Bayes is the one to go with as the implementation is rather straightforward and in most cases, it has similar performance as that of a logistic regression model.
2. Number of data points. In the era of big data it is common to encounter datasets with an enormous amount of samples. Since logistic regression is fit via an optimization routine, it can get very slow when the amount of data is large. **The Naive Bayes classifier does not have this problem as it only calculates means and variances and is extremely fast even for large datasets.**
3. **Number of covariates. As the number of covariates increases, the computational burden of logistic regression grows much faster than linearly. Naive Bayes does not have this problem (it scales linearly with the number of covariates) and should be considered as a strong candidate in this case.**

In our case, the dataset has less than 40,000 samples and only 9 covariates, so both logistic regression and Naive Bayes are OK.

Decision trees and random forests (40 mts)

Let's now discuss the **decision tree classifier**. The following figure shows an example. The nodes at the very bottom of the tree are called leaf nodes and all other nodes are called internal nodes or branch nodes. Each node contains a specific amount of samples that is indicated by `samples=` . Among these samples, the proportion of samples that are defaulters and non-defaulters are indicated by `value=` . The majority outcome values are indicated by `class=` .

When you want to predict the default status of a particular sample data point, you start with the top-most node. This internal node tells you which direction to go based on the value of `annual_inc` . Following the instructions in the internal nodes, you traverse the tree until you encounter a leaf node. The outcome label at that leaf node is then the predicted outcome for this sample. This whole prediction process is very much like what we do in our daily life when making a decision (based on the answer to a particular question, make a choice about what question to answer next and after enough answers, arrive at a conclusion).

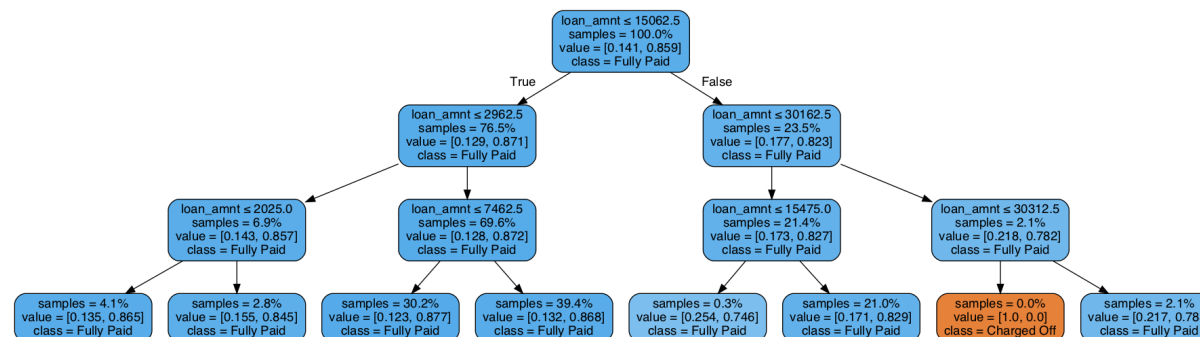
```
In [21]: clf = DecisionTreeClassifier(max_depth = 3)

# Train Decision Tree Classifier
clf = clf.fit(df[["loan_amnt"]], df.loan_status.cat.codes)

dot_data = StringIO()
export_graphviz(clf, out_file=dot_data, feature_names = ["loan_amnt"],
                class_names=['Charged Off', 'Fully Paid'],
                filled=True, rounded=True,
                special_characters=True,
                impurity = False, proportion = True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

Out[21]:



Exercise 11: (5 mts)

Assume you have two users with `annual_inc` equal to 10,000 and 100,000,000 respectively. What are the predicted default status for these two users, using the decision tree above?

Answer. For the first user, we go left, right, and right. The leaf node contains 2013 defaulters and 13229 non-defaulters. So we predict that this user will be a non-defaulter. For the second user, we go right, right, and right again. The predicted outcome for this user is also non-defaulter.

We can see the decision tree approach does not put any assumption on the relationship between the outcome and the covariates. Therefore, it is a very flexible algorithm that can capture a wide range of underlying relationships between the outcome and the covariates. Nevertheless, this algorithm has many drawbacks:

1. **Because it is so flexible, it tends to overfit.** When applied to an independent validation set, the performance often decreases significantly. This drawback really limits the application of decision trees.
2. **It is hard to train when the number of covariates are large.** At each internal node, you need to scan through a long list of potential splitting variables, thus making the algorithm computationally cumbersome. However, this issue has been mitigated with modern computing power.
3. Decision trees trained on the same dataset are not always the same. This instability issue happens because the algorithm may not scan through the covariates in the same order every time. **As a result, the final tree can vary a lot from run to run.**

Exercise 12: (10 mts)

Use `train_test_split` function from `sklearn` package, randomly select 75% of the sample in our dataset and train a decision tree on these samples with covariate `annual_inc`. Evaluate the accuracy of the tree on the training set and samples that are not included in the training set. Do you see a significant change in the accuracy score?

```
In [22]: X_train, X_test, y_train, y_test = train_test_split(df[["loan_amnt"]],
\
                                                         df.loan_status.cat.
codes, test_size = 0.25, \
                                                         random_state = 1)

clf_new = DecisionTreeClassifier()
clf_new = clf_new.fit(X_train, y_train)

y_pred_train = clf_new.predict(X_train)
y_pred_test = clf_new.predict(X_test)

print("Accuracy in training set:", accuracy_score(y_train, y_pred_train
))
print("Accuracy in the other samples:", accuracy_score(y_test, y_pred_t
est))
```

```
Accuracy in training set: 0.8617886178861789
Accuracy in the other samples: 0.8524336054562365
```

Answer. We can see the accuracy drops 0.1 in the other samples. This seems to indicate the overfitting problem of our particular decision tree is not too severe. But we should keep in mind that the prevalence of non-defaulters in the dataset is much higher than that of defaulters. This might inflate the accuracy of an otherwise mediocre classifier. (To see an example of this, consider a naive classifier which labels everyone a non-defaulter. This classifier would be right the vast majority of the time, yet we would hardly consider it to be a nuanced classifier.)

Decision boundary of a decision tree (5 mts)

The decision boundary of decision tree can be highly non-linear. It usually looks like patches due to the fact that we are dichotomizing the dataset at multiple levels. The following is an example for a tree built with `annual_inc` and `loan_amnt` :

```
In [23]: n_classes = 2
plot_colors = "br"

labels = ["Charged off", "Fully paid"]

clf = DecisionTreeClassifier().fit(df[["annual_inc", "loan_amnt"]], df.loan_status.cat.codes)

# Plot the decision boundary
plot_step = 100
viridis = cm.get_cmap('viridis', 2)
newcolors = viridis(np.linspace(0, 1, 2))
blue = np.array([0, 0, 1, 1])
red = np.array([1, 0, 0, 1])
newcolors[0, :] = blue
newcolors[1, :] = red
newcmp = ListedColormap(newcolors)

np.random.seed(0)
idx_plot = np.random.choice(df.shape[0], 100, replace = False )
df_plot = df.iloc[idx_plot]

x_min, x_max = df_plot.annual_inc.min() - 1000, df_plot.annual_inc.max() + 1000
y_min, y_max = df_plot.loan_amnt.min() - 1000, df_plot.loan_amnt.max() + 1000
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                     np.arange(y_min, y_max, plot_step))

fig, ax = plt.subplots()
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=newcmp, alpha = 0.7)
```

```

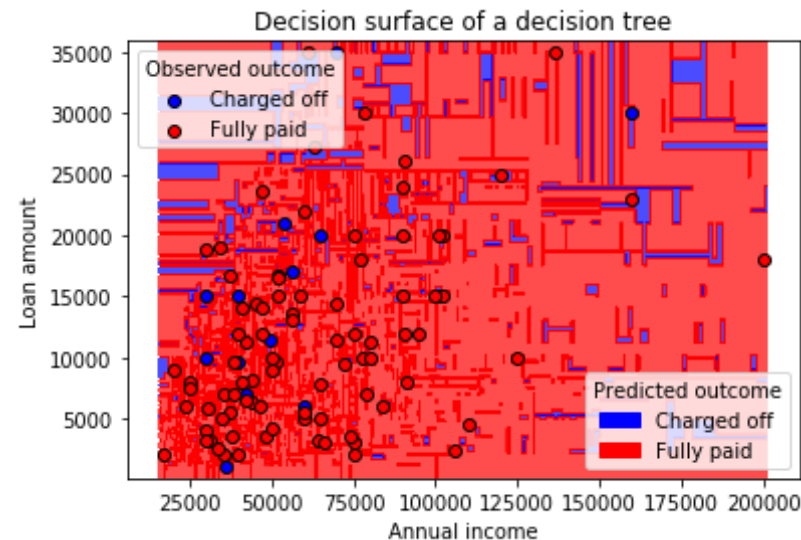
plt.xlabel("Annual income")
plt.ylabel("Loan amount")
plt.axis("tight");

for i, color in zip(range(n_classes), plot_colors):
    idx = np.where(df_plot.loan_status.cat.codes == i)
    plt.scatter(df_plot.iloc[idx].annual_inc, df_plot.iloc[idx].loan_ammnt, c=color,
                label=labels[i], edgecolors = "black")

legend_elements = [Patch(facecolor='blue', edgecolor='blue',
                          label='Charged off'),
                   Patch(facecolor='red', edgecolor='red',
                          label='Fully paid')]

plt.axis("tight")
plt.title("Decision surface of a decision tree")
leg1 = ax.legend(loc='upper left', title = "Observed outcome")
ax.legend(handles=legend_elements, loc = 'lower right', title = 'Predicted outcome');
ax.add_artist(leg1);

```



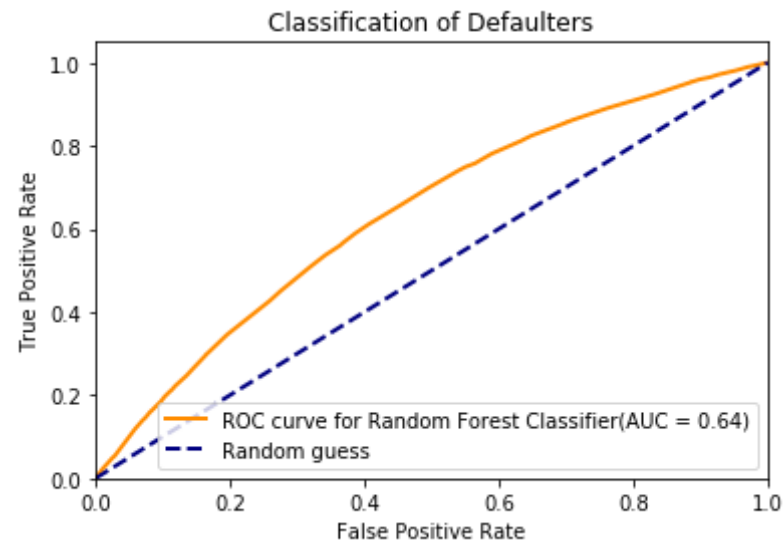
Random forest to improve the performance of decision tree (15 mts)

Random forests might well be the most popular machine learning algorithm for both classification and regression. The idea behind random forests is to construct a lot of decision trees, then use a voting mechanism to produce a final prediction that is agreed upon by most of the decision trees. When training these individual trees, we also subsample on the dataset as well as the covariates that are available in order to get as much built-in cross-validation as possible. As a result, the problem of overfitting and stability in a single decision tree model is resolved to a large degree. Let's fit a random forest classifier with covariates `term` and `annual_inc` and generate its ROC curve using the training data:

```
In [24]: clf_rf = RandomForestClassifier(n_estimators=100, max_depth=2, random_state=0)
clf_rf.fit( df_log2[['annual_inc', 'term']], df_loan_status.cat.codes )
clf_rf_probas = clf_rf.predict_proba(df_log2[['annual_inc', 'term']])

roc_p = roc_curve(df_loan_status.cat.codes, clf_rf_probas[:,1])
auc_p = auc_p = auc( roc_p[0], roc_p[1] )

plt.figure()
lw = 2
plt.plot(roc_p[0], roc_p[1], color='darkorange',
         lw=lw, label='ROC curve for Random Forest Classifier(AUC = %0.2f)' % auc_p)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--', label='Random guess')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Classification of Defaulters')
plt.legend(loc="lower right")
plt.show()
```



Exercise 13: (10 mts)

Use `cv_roc` as the template (same input arguments plus an argument `algo` determining decision tree or random forest, same output) to write a function `cv_roc_tree` that generates a 5-fold cross-validation ROC for a decision tree model and a random forest model with covariates `term` and `annual_inc`. Do you spot a significant increase in the prediction performance of the random forest model compared to the decision tree model?

```
In [25]: def cv_roc_tree(X, y, n_split, algo = 'Tree'):

    cv = StratifiedKFold(n_split)

    tprs = []
    aucs = []
    mean_fpr = np.linspace(0, 1, 100)
    for train, test in cv.split(X, y):
        if algo=="Tree":
```



```

        clf = DecisionTreeClassifier()
    else:
        clf = RandomForestClassifier(n_estimators=100, max_depth =
3, random_state=0)

    clf = clf.fit(X.iloc[train], y.iloc[train])
    probas_ = clf.predict_proba( X.iloc[test] )
    # Compute ROC curve and area the curve
    fpr, tpr, thresholds = roc_curve(y.iloc[test], probas_[:,1])
    tprs.append(interp(mean_fpr, fpr, tpr))
    tprs[-1][0] = 0.0
    roc_auc = auc(fpr, tpr)
    aucs.append(roc_auc)

    return tprs, aucs, mean_fpr

```

```

In [26]: tprs_t, aucs_t, mean_fpr_t = cv_roc_tree( df_log2[["term","annual_inc"
]], df_loan_status.cat.codes, 5)
tprs_f, aucs_f, mean_fpr_f = cv_roc_tree( df_log2[["term","annual_inc"
]], df_loan_status.cat.codes, 5, algo = "forest")

plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='blue',
        label='Random guess', alpha=.8)

mean_tpr_t = np.mean(tprs_t, axis=0)
mean_tpr_f = np.mean(tprs_f, axis=0)
mean_tpr_t[-1] = 1.0
mean_tpr_f[-1] = 1.0

mean_auc_t = auc(mean_fpr_t, mean_tpr_t)
mean_auc_f = auc(mean_fpr_f, mean_tpr_f)
std_auc_t = np.std(aucs_t)
std_auc_f = np.std(aucs_f)

plt.plot(mean_fpr_t, mean_tpr_t, color='darkorange',
        label=r'Decision Tree: Cross-validated Mean ROC (AUC = %0.2f
        $\pm$ %0.2f)' % (mean_auc_t, std_auc_t),
        lw=2, alpha=.8)
plt.plot(mean_fpr_f, mean_tpr_f, color='red',

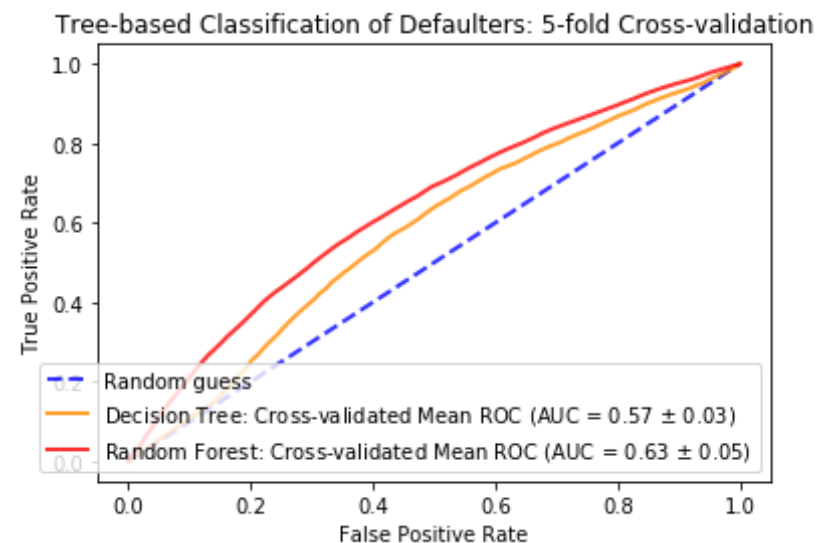
```

```

label=r'Random Forest: Cross-validated Mean ROC (AUC = %0.2f
$\pm$ %0.2f)' % (mean_auc_f, std_auc_f),
lw=2, alpha=.8)

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.title('Tree-based Classification of Defaulters: 5-fold Cross-validation')
plt.show()

```



Answer. We can see there is an increase in AUC using the random forest algorithm. With only two covariates, we can achieve a rather significant increase in the performance. This difference would likely be more pronounced with more variables.

Selecting the best model for predicting default status (10 mts)

So far we have learned about four different classification models, their strengths and weaknesses, and how to implement them. We are also equipped with a powerful model comparison tool, cross-validation, to get an objective evaluation for a classification model. Let's now construct all three models (we'll leave out decision tree for now, and just use random forest as both belong to the family of tree-based models) using all of our covariates. We want to compare their cross-validation ROC curves and AUC values to decide which should be the one that forms the user selection procedure for our company:

```
In [27]: tprs_nb, aucs_nb, mean_fpr_nb = cv_roc_nb( df.drop(["loan_status", "loan_amnt", "annual_inc"], axis=1), df.loan_status, 5)
tprs_f, aucs_f, mean_fpr_f = cv_roc_tree( df_log2.drop("Intercept", axis=1), df.loan_status.cat.codes, 5, algo = "forest")

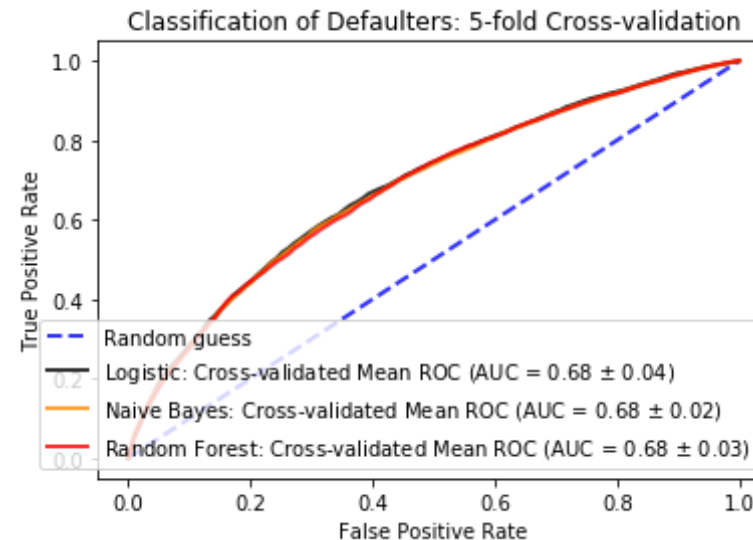
plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='blue',
         label='Random guess', alpha=.8)

mean_tpr_nb = np.mean(tprs_nb, axis=0)
mean_tpr_f = np.mean(tprs_f, axis=0)
mean_tpr_nb[-1] = 1.0
mean_tpr_f[-1] = 1.0

mean_auc_nb = auc(mean_fpr_nb, mean_tpr_nb)
mean_auc_f = auc(mean_fpr_f, mean_tpr_f)
std_auc_nb = np.std(aucs_nb)
std_auc_f = np.std(aucs_f)

plt.plot(mean_fpr_vs, mean_tpr_vs, color='black',
         label=r'Logistic: Cross-validated Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc_vs, std_auc_vs),
         lw=2, alpha=.8)
plt.plot(mean_fpr_nb, mean_tpr_nb, color='darkorange',
         label=r'Naive Bayes: Cross-validated Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc_nb, std_auc_nb),
         lw=2, alpha=.8)
plt.plot(mean_fpr_f, mean_tpr_f, color='red',
         label=r'Random Forest: Cross-validated Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc_f, std_auc_f),
         lw=2, alpha=.8)
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.title('Classification of Defaulters: 5-fold Cross-validation')
plt.show()
```



Based on the results above, we can pick either the logistic regression model or the Naive Bayes classifier as our final default detection model. In the future, if the amount of data required to be processed increases dramatically, we can prioritize the Naive Bayes model as a fast approximation to the logistic model. Sadly, random forest seems to have identical performance with longer training time. This is mainly because we didn't tune the parameters in the random forest algorithm, which is a challenge in and of itself. There are situations where random forest is clearly the best model, but that is beyond the scope of this case.

Conclusions (5 mts)

In this case, we learned about a few new classification models and the advantages and

disadvantages they have versus the standard logistic model. We learned that the Naive Bayes classifier, although "naive", generally does not suffer much and in fact is the fastest to train. On the flip side, tree-based models and particularly random forest are able to capture many non-linear relationships and interactions in our data, but are much slower to train. We then used cross-validation to select the model that has the best performance among the types of models we have learned so far.

Takeaways (5 mts)

Logistic regression, Naive Bayes, and tree-based models (decision trees and random forests) are all great classification models that each have their strengths and weaknesses. Cross-validation is an especially good way of evaluating the effectiveness of these models side by side.

In []: