case_11.1

May 29, 2020

1 How can I compare different models that predict the probability of defaulting on a loan?

[1]:
```
### Load relevant packages
import pandas                   as pd
import numpy                    as np
import matplotlib.pyplot        as plt
import seaborn                  as sns
import os

from scipy import stats

%matplotlib inline
plt.style.use('ggplot')

import warnings
warnings.filterwarnings('ignore')
```

1.1 Introduction (5 mts)

Business Context. Traditional commercial banks typically did not rely on statistical modeling to decide whether personal loans should be issued, although this is changing rapidly nowadays. You are a data scientist working in a modern commercial bank. Your data science team has already built simple regression models for predicting the probability that those loans would be defaulted on. However, you have noticed that many of these models perform much worse in production than they do in testing.

Business Problem. Your task is to **build a default probability model that you feel comfortable putting into production.**

Analytical Context. The dataset contains the details of 5000 loans requests that have been previously issued by your bank. For each loan, the final status of the loan (i.e. whether the loan defaulted) is also available:

1. The file **"loan_light.csv"** contains the details of 5000 loans
2. The file **"loan_param.xlsx"** contains the description of each covariate

The case will proceed as follows: you will 1) perform some data exploration to determine the appropriate variable transformations to make; 2) fit some simple models; 3) learn about **cross-validation** and use this to select the best simple model; and finally 4) responsibly construct more complex models using cross-validation.

1.2 Data exploration (40 mts)

Let's start by taking a look at the data:

[2]:
```
Data = pd.read_csv("loan_light.csv")
Data = Data.sample(frac=1)  #shuffle the rows
```

[3]:
```
Data.head()
```

[3]:

	annual_inc	application_type	avg_cur_bal	chargeoff_within_12_mths	\
3394	145000.0	Individual	4087.0	0.0	
3206	100000.0	Individual	5016.0	0.0	
4462	50000.0	Individual	1792.0	0.0	
1093	75000.0	Individual	11702.0	0.0	
1555	85000.0	Individual	22524.0	0.0	

	delinq_2yrs	dti	emp_length	grade	inq_last_12m	installment	…	\
3394	0.0	6.21	10	B	4.0	408.31	…	
3206	2.0	30.07	1	C	0.0	254.89	…	
4462	0.0	16.19	6	C	3.0	483.74	…	
1093	1.0	37.70	6	D	4.0	553.87	…	
1555	0.0	24.87	4	A	4.0	301.15	…	

	num_actv_bc_tl	pub_rec_bankruptcies	home_ownership	term	mort_acc	\
3394	1.0	2.0	RENT	36	0.0	
3206	4.0	0.0	OWN	36	0.0	
4462	6.0	0.0	RENT	36	0.0	
1093	3.0	0.0	OWN	36	0.0	
1555	1.0	0.0	MORTGAGE	36	2.0	

	num_tl_90g_dpd_24m	purpose	year	loan_default	job
3394	0.0	home_improvement	2016	0	owner
3206	0.0	home_improvement	2016	1	practitioner
4462	0.0	debt_consolidation	2016	1	assistant
1093	0.0	credit_card	2016	0	manager
1555	0.0	home_improvement	2017	0	manager

[5 rows x 21 columns]

[4]:
```
Data.keys()
```

[4]:
```
Index(['annual_inc', 'application_type', 'avg_cur_bal',
       'chargeoff_within_12_mths', 'delinq_2yrs', 'dti', 'emp_length', 'grade',
       'inq_last_12m', 'installment', 'loan_amnt', 'num_actv_bc_tl',
       'pub_rec_bankruptcies', 'home_ownership', 'term', 'mort_acc',
       'num_tl_90g_dpd_24m', 'purpose', 'year', 'loan_default', 'job'],
      dtype='object')
```

[5]:
```
df_description = pd.read_excel('loan_param.xlsx').dropna()
df_description.style.set_properties(subset=['Description'], **{'width':␣
↪'1000px'})
```

[5]:
```
<pandas.io.formats.style.Styler at 0x1a17c96b00>
```

1.2.1 Exercise 1: (20 mts)

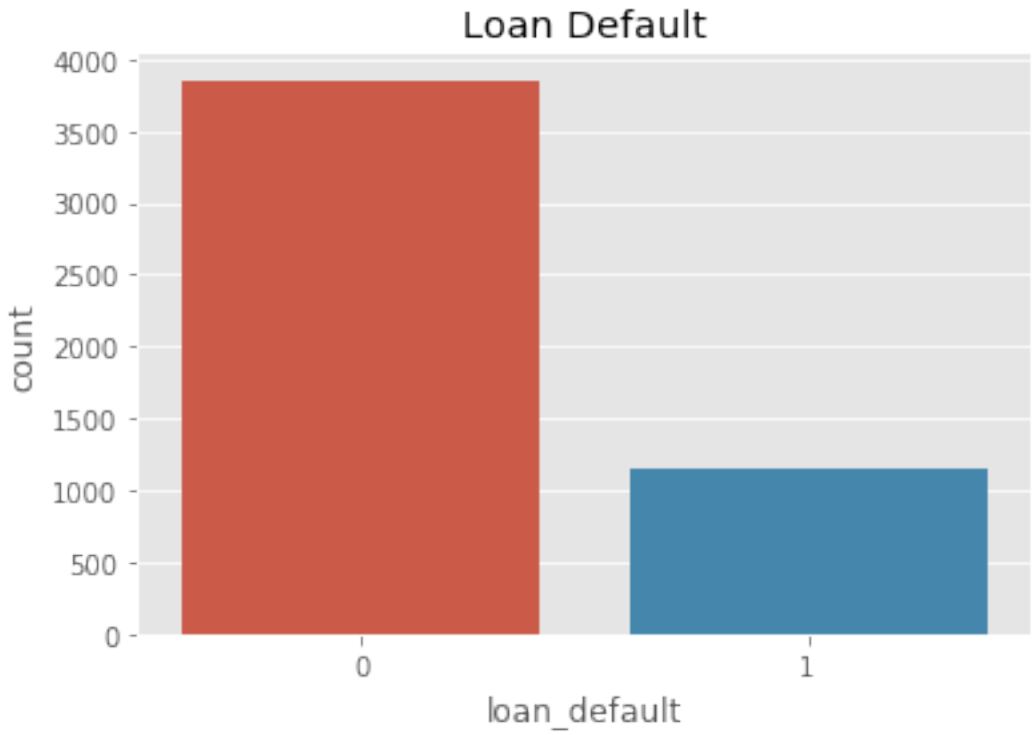
For each of the following, perform the directed visualization and discuss your conclusions from it.

1.1 Create a bar chart showing the number of loans that did and did not default.

Answer. One possible solution is shown below:

[6]:
```
sns.countplot(x='loan_default', data = Data)
plt.title("Loan Default")
```

[6]:
```
Text(0.5, 1.0, 'Loan Default')
```



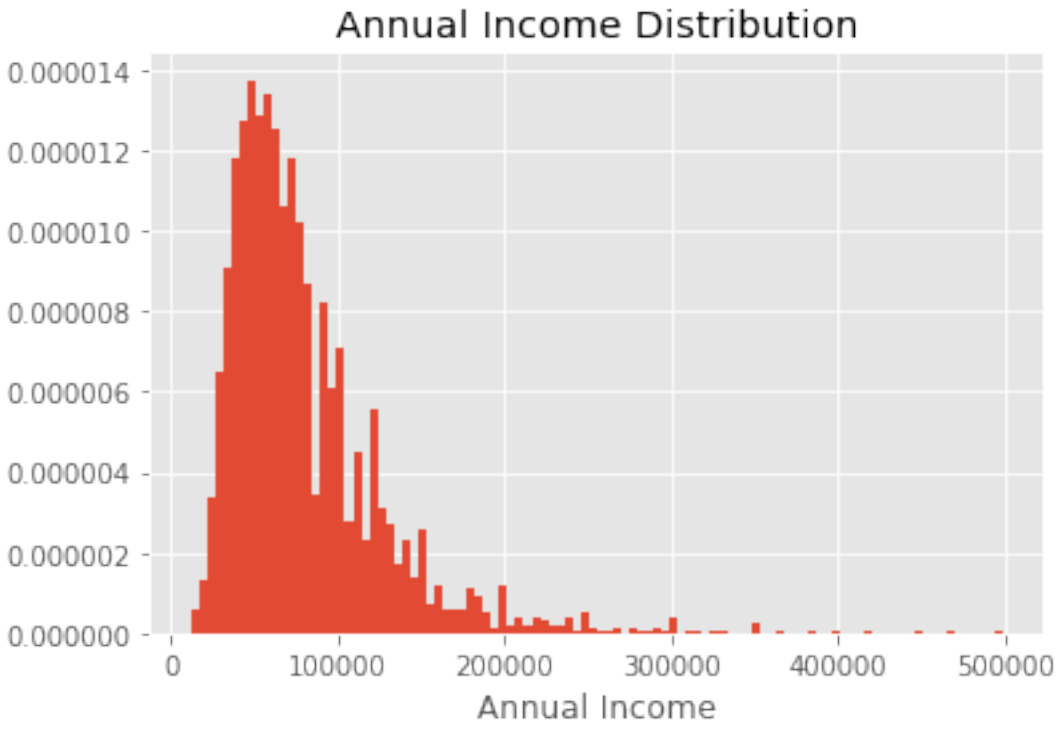
We see that around 20 - 25 percent of all loans in the dataset defaulted.

1.2 Plot a histogram of the annual incomes.

Answer. One possible solution is given below:

[7]:
```
Data.annual_inc.hist(bins=100, density=True)
plt.title("Annual Income Distribution")
plt.xlabel("Annual Income")
```

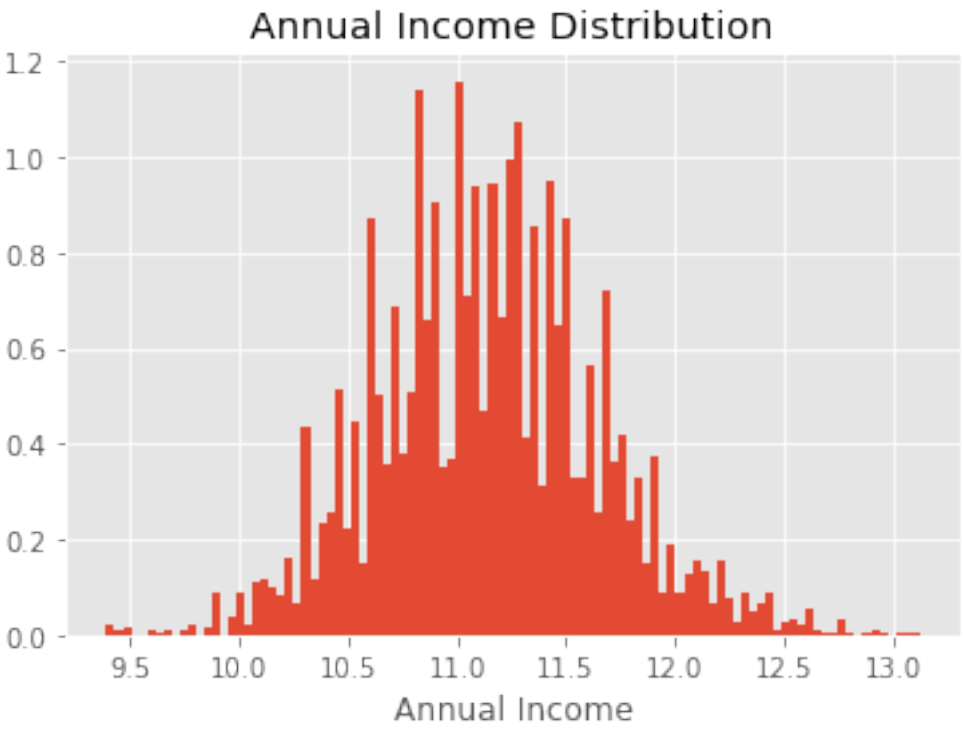
[7]:
```
Text(0.5, 0, 'Annual Income')
```



We see that the data is quite skewed. Let's try a logarithmic transformation to get the data to be more normally distributed:

[8]:
```
np.log(Data.annual_inc).hist(bins=100, density=True)
plt.title("Annual Income Distribution")
plt.xlabel("Annual Income")
```

[8]:
```
Text(0.5, 0, 'Annual Income')
```

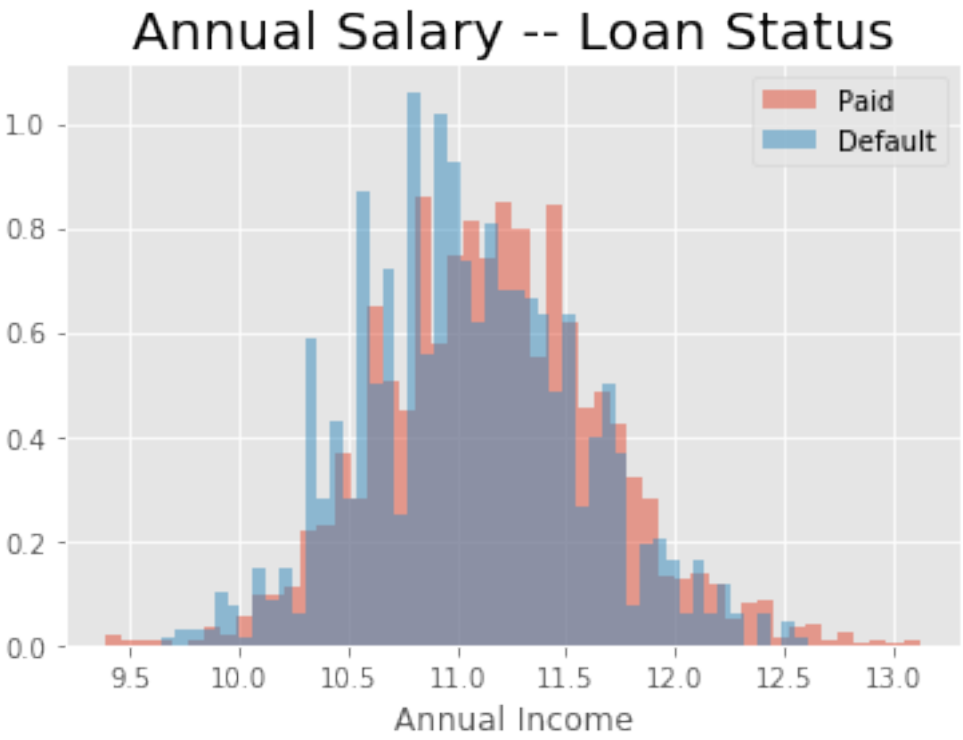


1.3 Is the distribution of annual incomes different between applicants who defaulted vs. applicants who did not default on their loans?

Answer. One possible solution is given below:

[9]:
```
np.log(Data['annual_inc'][Data.loan_default == 0]).hist(bins=50, density=True,␣
↪alpha=0.5, label="Paid")
np.log(Data['annual_inc'][Data.loan_default == 1]).hist(bins=50, density=True,␣
↪alpha=0.5, label="Default")
plt.xlabel("Annual Income")
plt.legend()
plt.title("Annual Salary -- Loan Status", fontsize=20)
```

[9]:
```
Text(0.5, 1.0, 'Annual Salary -- Loan Status')
```



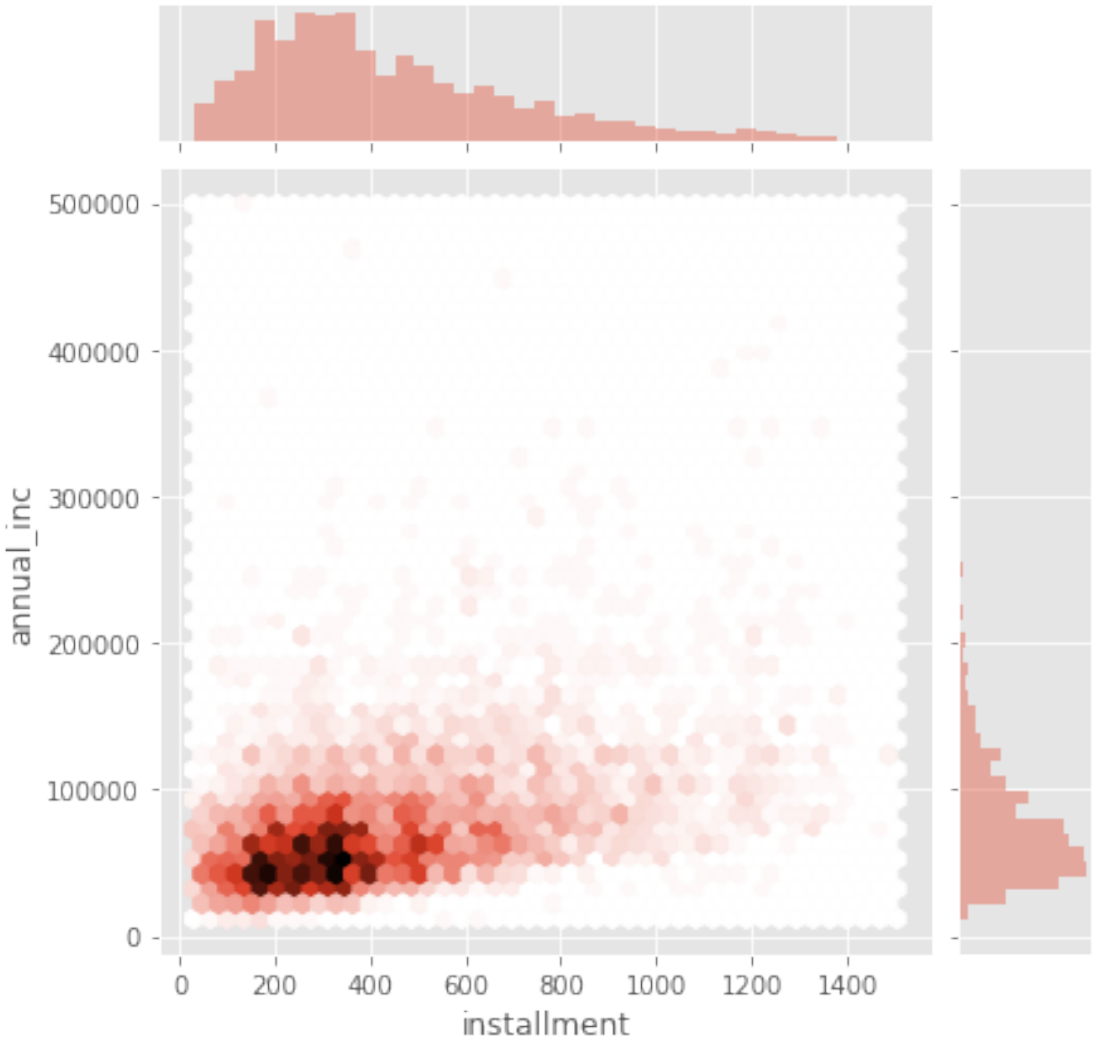
We can see that the distributions are not that different, indicating that income alone is not likely to explain a significant fraction of the difference in loan default status.

1.4 Explore the association between annual income and the monthly installment.

Answer. One possible solution is below:

[10]:
```
sns.jointplot(Data.installment, Data.annual_inc, kind="hex")
```

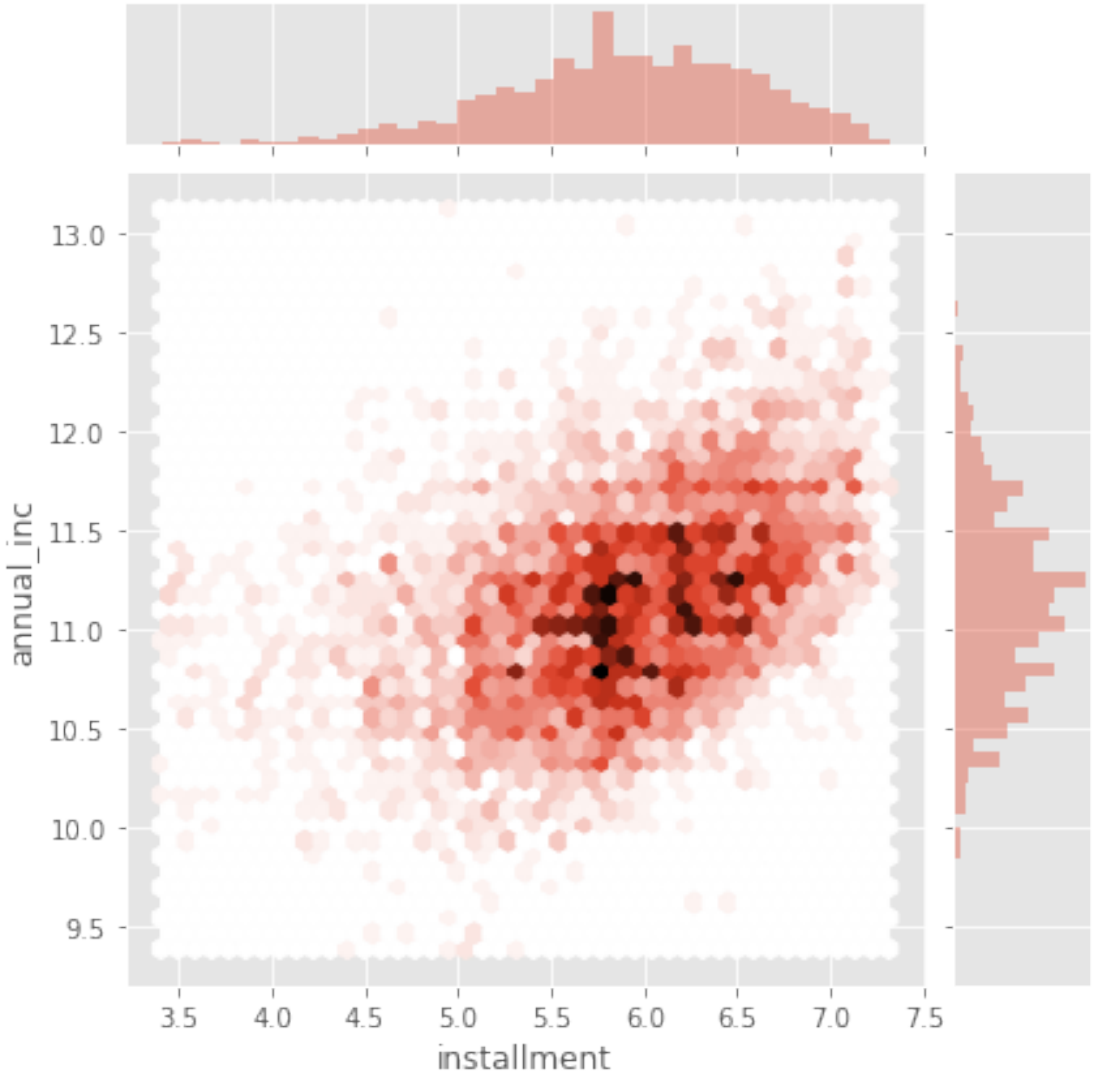
[10]:
```
<seaborn.axisgrid.JointGrid at 0x1a186f8ba8>
```



This shows that the distributions for both variables are skewed in similar ways. This means both ought to undergo a log transform:

[11]:
```
sns.jointplot(np.log(Data.installment), np.log(Data.annual_inc), kind="hex")
```

[11]:
```
<seaborn.axisgrid.JointGrid at 0x1a18b18dd8>
```



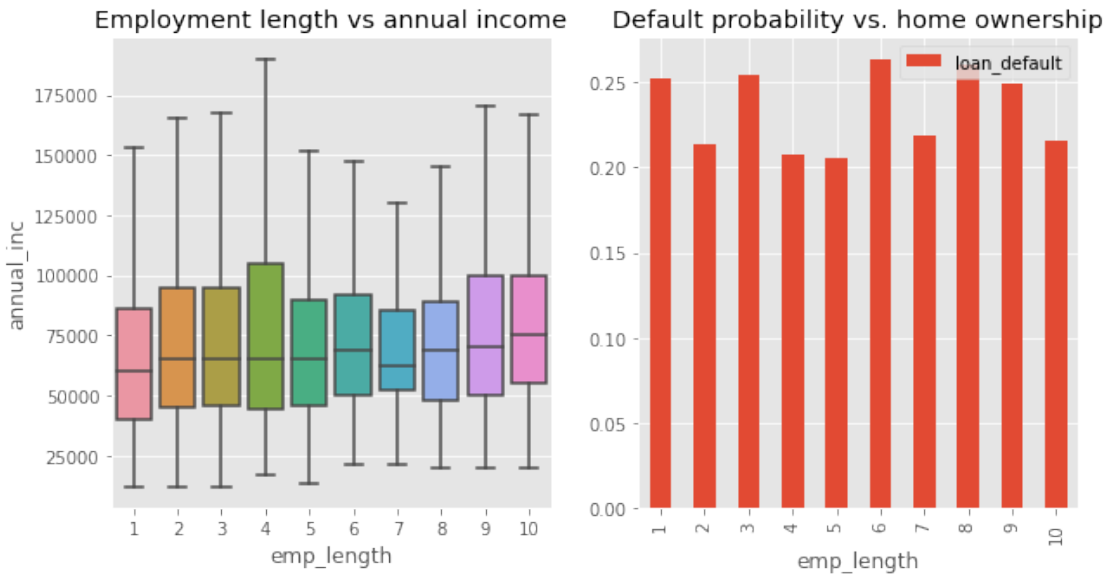
This looks much better and seems to suggest (with some outliers on the far left) a linear relationship between the logarithms of both variables.

Here are a few more figures which look at the relationship between other numerical covariates and the probability of default, as well as annual income:

`emp_length`:

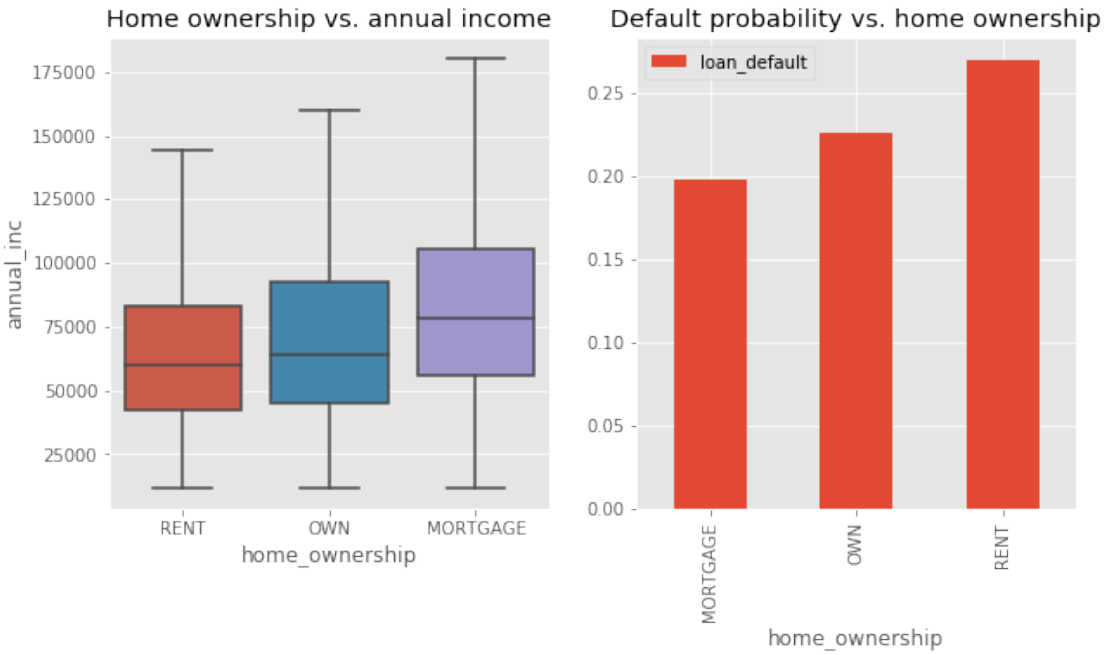
[12]:
```
fig, (ax1, ax2) = plt.subplots(figsize = (10,5), ncols=2, sharey= False)
sns.boxplot(x='emp_length', y = 'annual_inc', data = Data, showfliers=False, ax␣
↪= ax1) #showfliers=False for nice display
ax1.set_title("Employment length vs annual income")
Data[["emp_length",'loan_default']].groupby("emp_length").mean().plot.
↪bar(rot=90,ax = ax2)
```

```
plt.title("Default probability vs. home ownership");
```



homeOwnership:

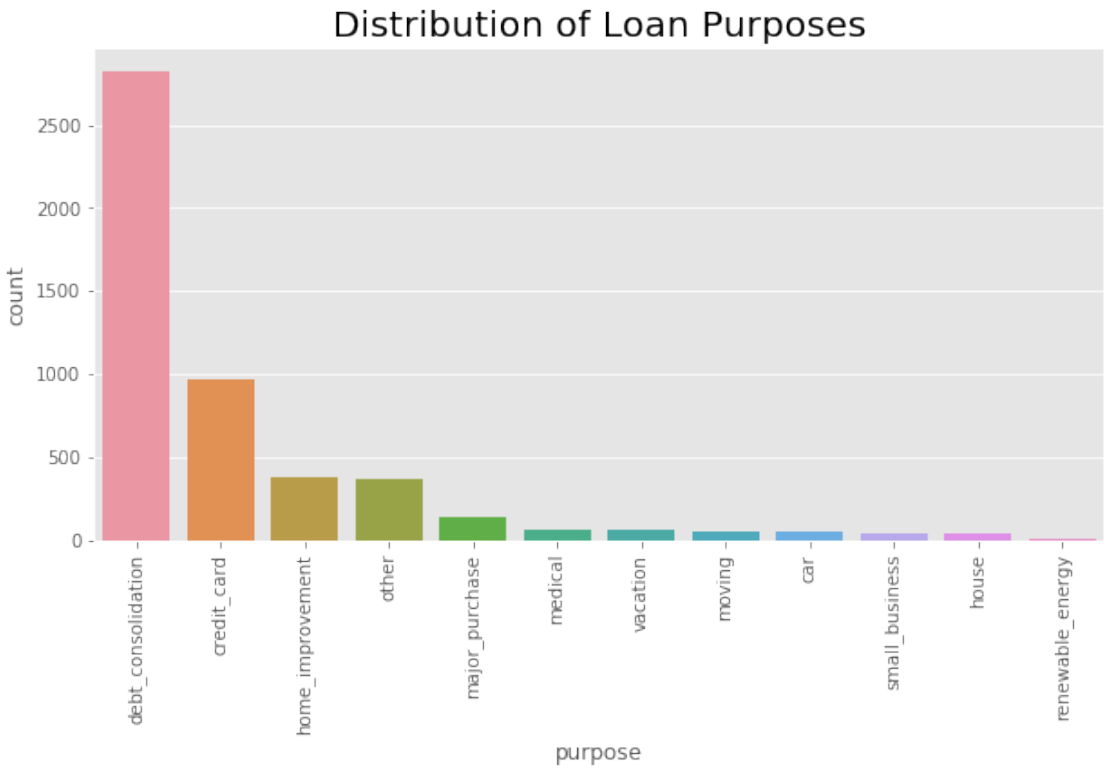
[13]:
```
fig, (ax1, ax2) = plt.subplots(figsize = (10,5), ncols=2, sharey= False)
sns.boxplot(x="home_ownership",y="annual_inc", data = Data, showfliers=False,␣
 ↪ax = ax1) #showfliers=False for nice display
ax1.set_title("Home ownership vs. annual income")
Data[["home_ownership",'loan_default']].groupby("home_ownership").mean().plot.
 ↪bar(rot=90,ax = ax2)
plt.title("Default probability vs. home ownership");
```



Here are some figures that show the relationship between various categorical variables and the probability of default:

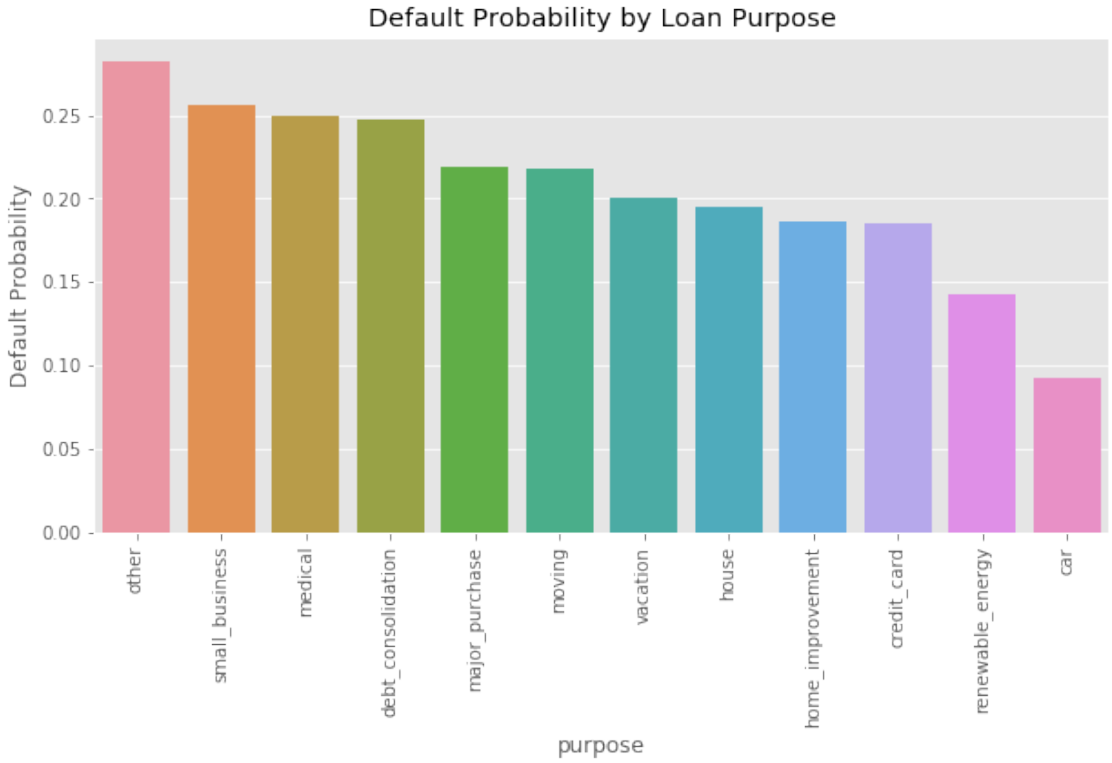
purpose:

[14]:
```
plt.figure(figsize= (10,5))
Data.emp_length.value_counts()
sns.countplot(x='purpose', order=Data['purpose'].value_counts().index, data =␣
 ↪Data)
plt.xticks(rotation=90)
plt.title("Distribution of Loan Purposes", fontsize=20);
```



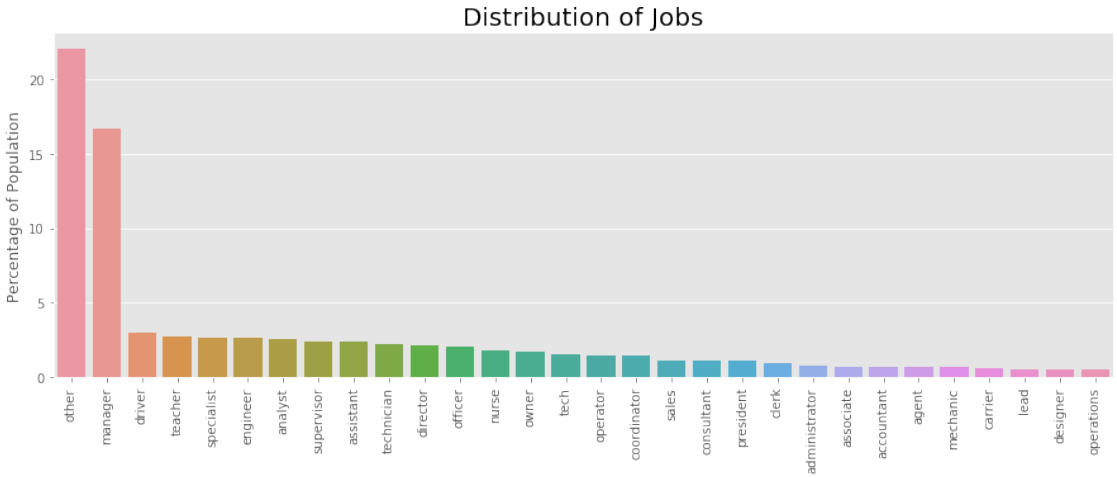
[15]:
```
plt.figure(figsize= (10,5))
purpose_default = Data[["loan_default", "purpose"]].groupby("purpose").mean()
purpose_default = purpose_default.sort_values(by="loan_default",axis=0,␣
 ↪ascending=False)
sns.barplot(x=purpose_default.index[:30],
            y=purpose_default["loan_default"][:30].values,
            orient="v")
plt.xticks(rotation=90)
plt.ylabel("Default Probability");
plt.title("Default Probability by Loan Purpose")
```

[15]:
```
Text(0.5, 1.0, 'Default Probability by Loan Purpose')
```

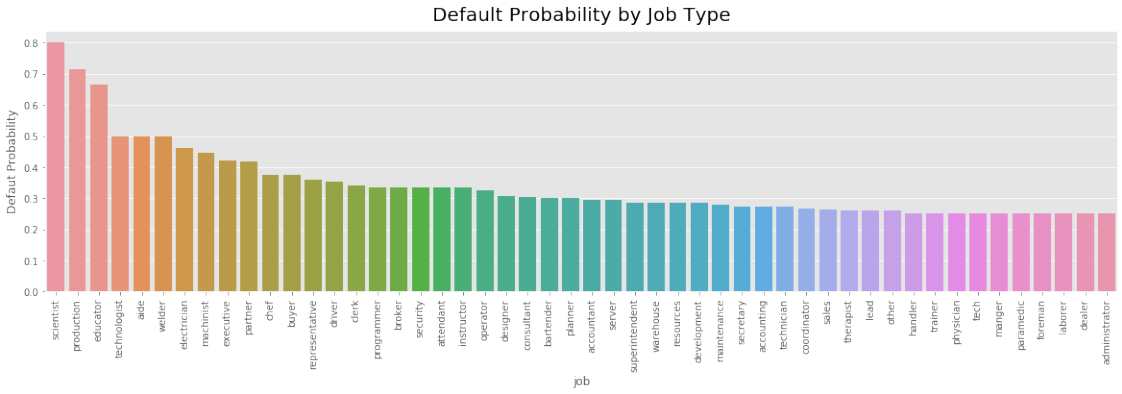


job:

[16]:
```
plt.figure(figsize= (15,5))
sns.barplot(x=Data["job"].value_counts()[:30].index.values ,
            y=100 * Data.job.value_counts()[:30].values / len(Data),
            orient="v")
plt.xticks(rotation=90)
plt.ylabel("Percentage of Population")
plt.title("Distribution of Jobs", fontsize=20);
```



[17]:
```
plt.figure(figsize= (20,5))

df_job_default = Data[["loan_default", "job"]].groupby("job").mean()
df_job_default = df_job_default.sort_values(by="loan_default",axis=0,␣
 ↪ascending=False)
sns.barplot(x=df_job_default.index[:50],
            y=df_job_default["loan_default"][:50].values,
            orient="v")
plt.xticks(rotation=90)

plt.ylabel("Defaut Probability")
plt.title("Default Probability by Job Type", fontsize=20,␣
 ↪verticalalignment='bottom');
```

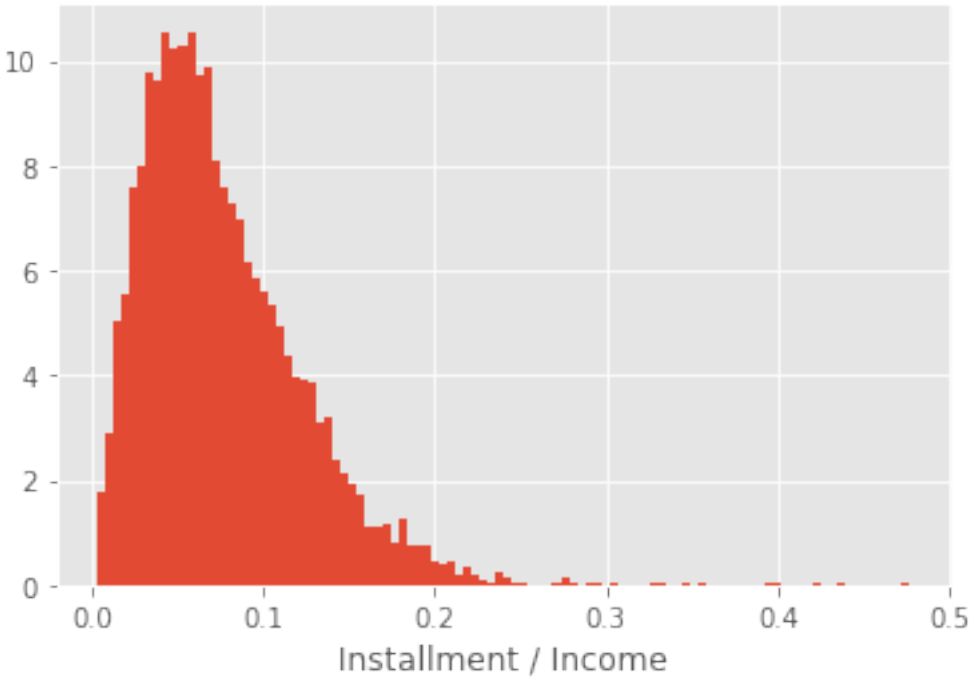


1.2.2 Adding a new variable

The yearly payment owed by the borrower, as a fraction of their annual income, is a standard metric used in evaluating whether a loan should be issued. Let's define a new variable **"install_income"** which codes the installment as a fraction of the annual income and study its association with the other features:

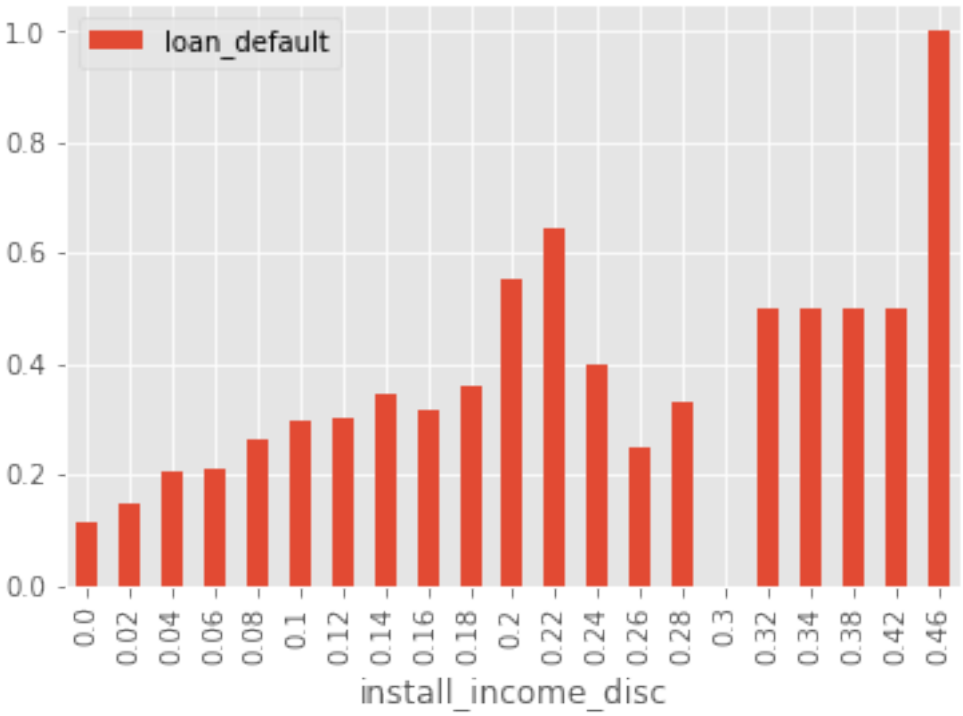
[18]:
```
Data['install_income'] = 12 * Data.installment / Data.annual_inc
H = plt.hist(Data['install_income'], bins=100, density=True)
plt.xlabel(r"Installment / Income")
```

[18]:
```
Text(0.5, 0, 'Installment / Income')
```



In order to easily investigate this variable's association with the probability of default, define a new covariate named `install_income_disc` that is a discretized version of `install_income`:

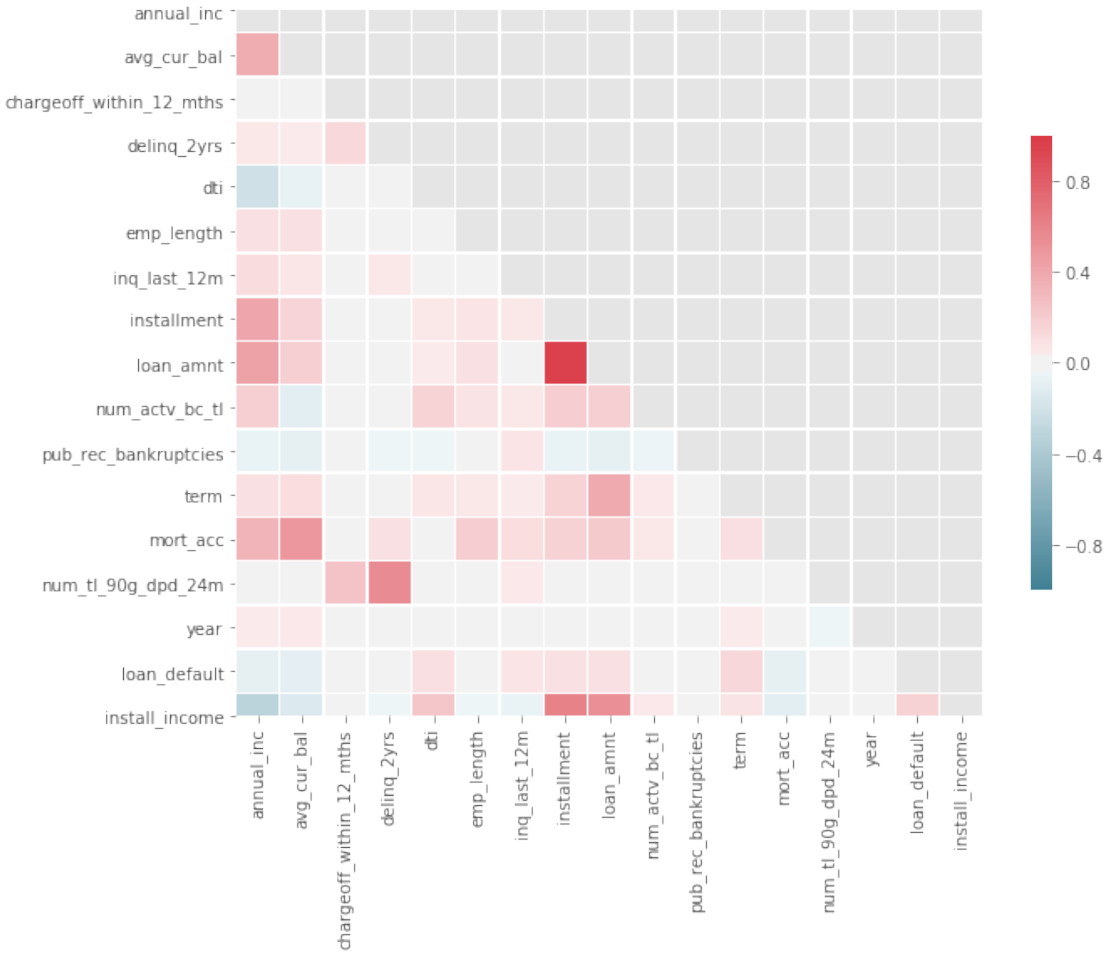
[19]:
```
# let us discretize the "install_income" variable to study the probability of␣
 ↪default
# as a function of "install_income"
Data["install_income_disc"] = (Data.install_income*50).astype(int)/50. ␣
 ↪#discretization
Data[["loan_default", "install_income_disc"]].groupby("install_income_disc").
 ↪mean().plot.bar(rot=90)
Data = Data.drop(["install_income_disc"], axis=1)

# --> there is a clear positive association: as the fraction of the annual␣
 ↪income devoted to the re-imbursement of
# the loan increases, the probability of default sharply increases
```



1.2.3 Exercise 2: (10 mts)

Visualize the correlation matrix across all numerical features by using the `sns.heatmap()` command:

[20]:
```
#compute correlation matrix
df_correlations = Data.corr()

#mask the upper half for visualization purposes
mask = np.zeros_like(df_correlations, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Draw the heatmap with the mask and correct aspect ratio
plt.figure(figsize= (10,10))

cmap = sns.diverging_palette(220, 10, as_cmap=True)
sns.heatmap(df_correlations,mask=mask,  vmax=1, vmin=-1, cmap=cmap,
            center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5});
```



1.3 Building a predictive model (20 mts)

Let's first start by building a standard logistic regression model. In general, it is important and extremely useful to first create baseline/simple models which can be compared to more complex models later.

1.3.1 Exercise 3: (15 mts)

3.1 Using the `LogisticRegression()` function from `scikit-learn`, write a function named `fit_logistic_regression(X,y)` that fits a logistic regression on the array of covariates `X` and associated response variable `y`.

Answer. One possible solution is shown below:

[21]:
```
from sklearn.linear_model import LogisticRegression
def fit_logistic_regression(X,y):
```

```
    """
    fit a logistic regression with feature matrix X and binary output y
    """
    clf = LogisticRegression(solver='lbfgs', tol=10**-4,
                             fit_intercept=True,
                             multi_class='multinomial').fit(X,y)
    return clf
```

3.2 Create a basic logistic regression model for predicting the loan default with only one feature: `install_income`. Call this model `model1`. Use a 70/30 train-test split of the data.

Answer. One possible solution is shown below:

[22]:
```
# we will use a 70%/30% split for training/validation
n_total = len(Data)
n_train = int(0.7*n_total)

X, y = Data[["install_income"]], Data.loan_default
X_train, y_train = X[:n_train], y[:n_train]
X_test, y_test = X[n_train:], y[n_train:]
```

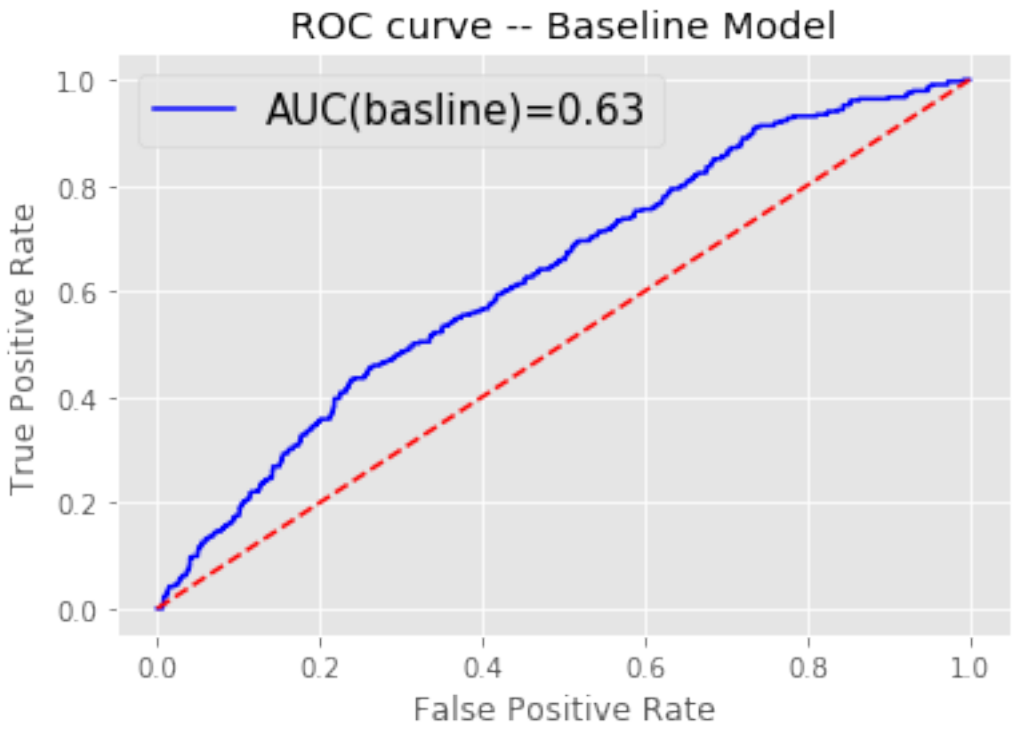
[23]:
```
model1 = fit_logistic_regression(X_train, y_train)  # fit a logistic regression
y_test_pred = model1.predict_proba(X_test)[:,1]     # make probabilistic␣
 ↪predictions on test set
```

3.3 Plot the ROC curve of `model1` and find the area under the curve.

Answer. One possible solution is shown below:

[24]:
```
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import StratifiedKFold
```

[25]:
```
fpr, tpr, _ = roc_curve(y_test, y_test_pred)  #compute FPR/TPR
auc_baseline = auc(fpr, tpr) # compute AUC

plt.plot(fpr, tpr, "b-", label="AUC(basline)={:2.2f}".format(auc_baseline))
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(fontsize=15)
plt.plot([0,1], [0,1], "r--")
plt.title("ROC curve -- Baseline Model");
```



1.3.2 Exercise 4: (5 mts)

4.1 Consider `model1` from above. Would you want this to be your final model? Why or why not?

Answer. This should not be the final model. This is because we have not explored the contribution from other variables, which in addition to containing valuable information could also be confounding the perceived effect of `install_income` on the response variable. This under-exploitation of information is called **underfitting**.

4.2 Let's instead put all the variables available in the model, so that we are maximally leveraging our available info. Would you be in favor of this or not?

Answer. This is also a bad idea. If we *blindly* use all of the variables in our model fitting, a phenomenon called **overfitting** occurs. This is when a statistical model "fits" too closely to a particular set of data, which may well be noisy and exhibit randomness and therefore fail to predict future, different observations reliably.

1.4 Cross-validation (30 mts)

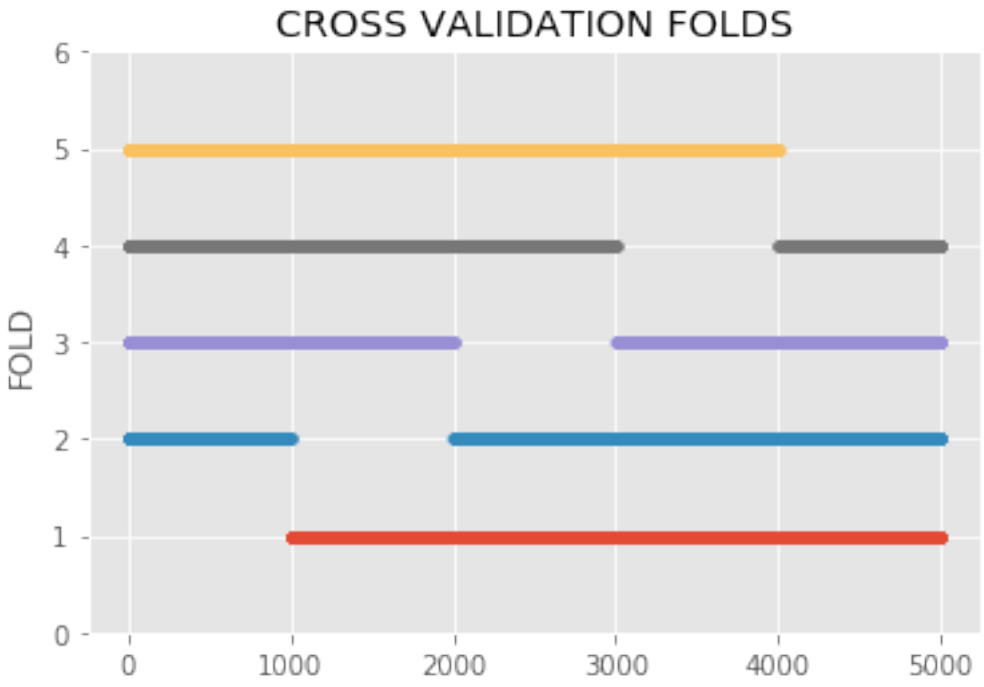
Cross-validation is a set of techniques for assessing how well the results of a model will generalize to an out-of-sample dataset; i.e. in practice or production. It is chiefly used to flag overfitting.

Cross-validation works as follows: one splits the available data into k sets, or **folds**. $k-1$ of these folds will be used to train the model, while the held-out fold will be used as the test set on which the model is evaluated. For computational stability, this procedure is generally split many times, such that each fold has an opportunity to serve as the test set. For each repetition, a metric of prediction performance (e.g. AUC) is calculated on the test set. The average of these metrics, as well as their standard deviation, is then reported. An example is shown here for 5-fold cross-validation:

Let's do this with code. The following code displays the 5 different folds used in a standard 5-fold cross-validation approach. To do so, use the `StratifiedKFold()` function from `scikit-learn`:

[26]:
```
skf = StratifiedKFold(n_splits=5)
for k, (train_index, test_index) in enumerate( skf.split(X, y) ):
    plt.plot(train_index, [k+1 for _ in train_index], ".")
plt.ylim(0,6)
plt.ylabel("FOLD")
plt.title("CROSS VALIDATION FOLDS")
```

[26]:
```
Text(0.5, 1.0, 'CROSS VALIDATION FOLDS')
```



The following code defines a function `compute_AUC(X, y, train_index, test_index)` that computes the AUC of a model trained on "train_index" and tested in "test_index".

[27]:
```
def compute_AUC(X, y, train_index, test_index):
    """
    feature/output: X, y
    dataset split: train_index, test_index
    """
    X_train, y_train = X.iloc[train_index], y.iloc[train_index]
    X_test, y_test = X.iloc[test_index], y.iloc[test_index]

    clf = fit_logistic_regression(X_train, y_train)
    default_proba_test = clf.predict_proba(X_test)[:,1]
    fpr, tpr, _ = roc_curve(y_test, default_proba_test)
    auc_score = auc(fpr, tpr)
    return auc_score, fpr, tpr
```

1.4.1 Exercise 5: (5 mts)

With the help of the `compute_AUC` function defined above, write a function `cross_validation_AUC(X,y,nfold)` that carries out a 10-fold cross-validation and returns a list which contains the area under the curve for each fold of the cross-validation.

Answer. One possible solution is shown below:

[28]:
```
def cross_validation_AUC(X,y, nfold=10):
    """
    use a n-fold cross-validation for computing AUC estimates
    """
    skf = StratifiedKFold(n_splits=nfold)  #create a cross-validation splitting
    auc_list = [] #this list will contain the AUC estimates associated with␣
 ↪each fold
    for k, (train_index, test_index) in enumerate( skf.split(X, y) ):
        auc_score, _, _ = compute_AUC(X, y, train_index, test_index)
        auc_list.append(auc_score)
    return auc_list
```

We will now estimate and compare, through cross-validation analysis, the performance of all the "simple models" that only use one numerical feature as input. As discussed in the EDA section, we will use the logarithmic transform for the `anual_income`, `loan_amount`, and `avg_cur_bal` variables:

[29]:
```
# let us extract only the numerical (i.e non-categorical) features
numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
Data_numerics = Data.select_dtypes(include=numerics)
Data_numerics = Data_numerics.drop(["installment", "year"], axis=1)
```

```
# Using a log scale when appropriate
Data_numerics["annual_inc"] = np.log10(Data_numerics["annual_inc"])
Data_numerics["loan_amnt"] = np.log10(Data_numerics["loan_amnt"])
Data_numerics["avg_cur_bal"] = np.log10(1.+Data_numerics["avg_cur_bal"])
```

Let's compute cross-validation estimates of the AUC for each single-feature model:

[30]:
```
model_perf = pd.DataFrame({}) #this data-frame will contain the AUC estimates
for key in Data_numerics.keys():
    if key == "loan_default": continue
    X_full, y_full = Data_numerics[[key]], Data_numerics.loan_default
    auc_list = cross_validation_AUC(X_full, y_full, nfold=10)
    model_perf["SIMPLE:" + key] = auc_list
```

1.4.2 Exercise 6: (5 mts)

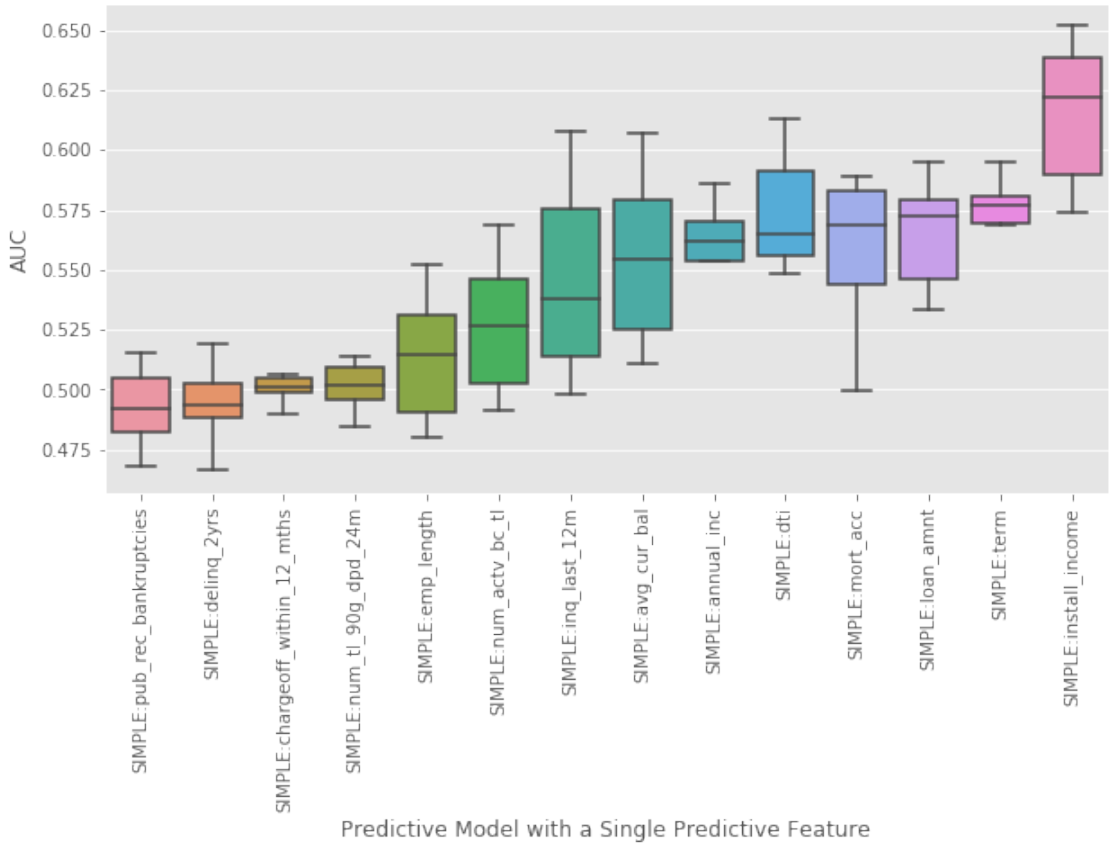
Construct a boxplot which shows the distribution of cross-validation scores of each variable (remember, each variable has 10 total scores). Which feature has the highest/lowest predictive power?

Answer. One possible solution is shown below:

[31]:
```
def plot_boxplot_ordered(df_model):
    """
    display a list of boxplot, ordered by the media values
    """
    df = df_model[df_model.median().sort_values().index]
    sns.boxplot(x="variable", y="value", data=pd.melt(df), showfliers=False)
    plt.xticks(rotation=90)
```

[32]:
```
plt.figure(figsize= (10,5))
plot_boxplot_ordered(model_perf)
plt.xlabel("Predictive Model with a Single Predictive Feature")
plt.ylabel("AUC")
```

[32]:
```
Text(0, 0.5, 'AUC')
```



1.4.3 Exercise 7: (5 mts)

Consider the model that consists of using *all* the numerical features (and none of the categorical features). Carry out a 10-fold cross-validation analysis to determine whether this model has better predictive performance than the best single-feature model. Use the boxplot method again as we did in Exercise 7.

Answer. One possible solution is shown below:

[33]:
```
X_full, y_full = Data_numerics.drop(["loan_default"], axis=1), Data_numerics.
 ↪loan_default
auc_list = cross_validation_AUC(X_full, y_full)
model_perf["ALL_NUMERICAL"] = auc_list
model_perf
```

[33]:

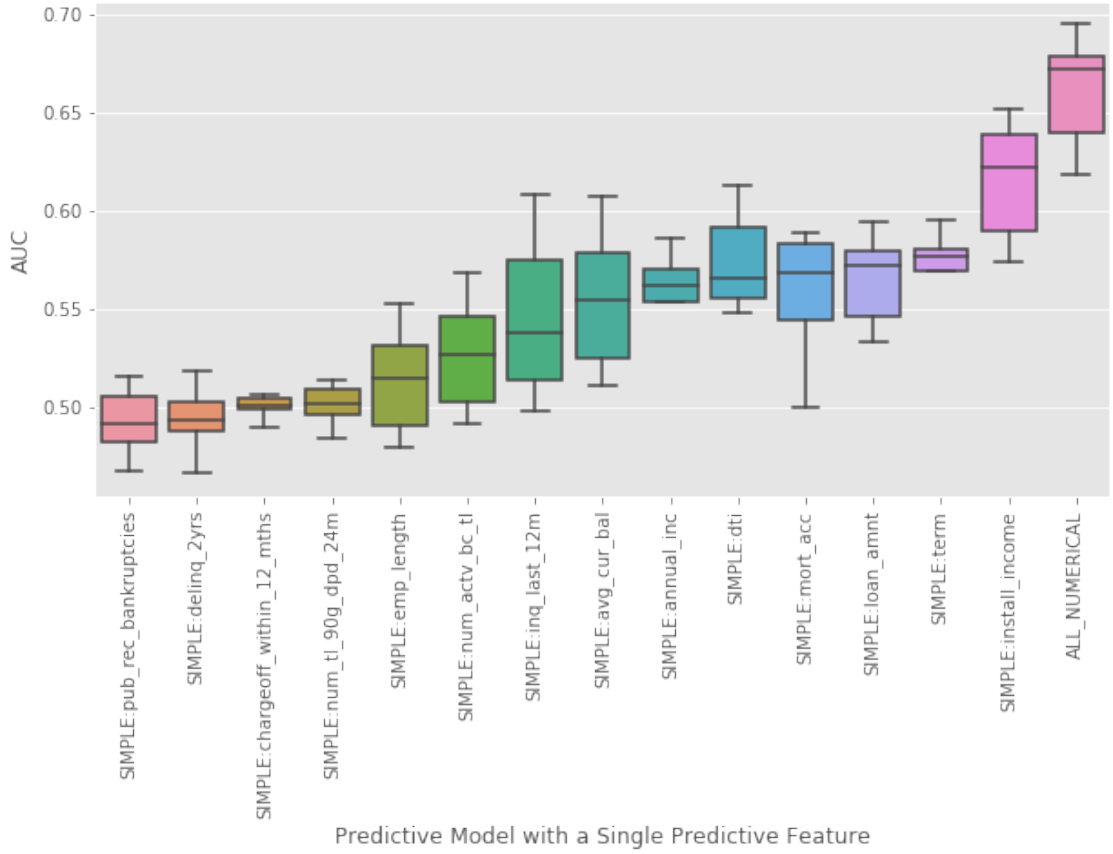
	SIMPLE:annual_inc	SIMPLE:avg_cur_bal	SIMPLE:chargeoff_within_12_mths	\
0	0.569362	0.545393	0.499099	
1	0.569903	0.607209	0.491304	
2	0.504776	0.523868	0.503435	
3	0.594568	0.583783	0.498250	

```
4           0.554376            0.581762                          0.489554
5           0.529249            0.569848                          0.499560
6           0.586008            0.562891                          0.505195
7           0.553566            0.510492                          0.502597
8           0.568364            0.516951                          0.505195
9           0.555081            0.527820                          0.506494

   SIMPLE:delinq_2yrs  SIMPLE:dti  SIMPLE:emp_length  SIMPLE:inq_last_12m  \
0            0.498355    0.612818           0.487700             0.512120
1            0.518732    0.565386           0.479815             0.509946
2            0.500338    0.548040           0.552399             0.577349
3            0.507352    0.589509           0.516194             0.598419
4            0.488120    0.555573           0.513066             0.517448
5            0.466121    0.555720           0.549904             0.607849
6            0.487883    0.553586           0.525353             0.498148
7            0.503281    0.612474           0.500353             0.536067
8            0.488574    0.565026           0.533094             0.568341
9            0.472306    0.591969           0.485771             0.539155

   SIMPLE:loan_amnt  SIMPLE:num_actv_bc_tl  SIMPLE:pub_rec_bankruptcies  \
0          0.485425               0.549403                     0.515139
1          0.584062               0.553909                     0.507952
2          0.572156               0.500631                     0.484535
3          0.533179               0.535573                     0.495110
4          0.580553               0.501558                     0.481536
5          0.569949               0.523557                     0.467916
6          0.572513               0.505353                     0.488809
7          0.575985               0.568751                     0.497448
8          0.538585               0.491479                     0.469105
9          0.594577               0.529916                     0.507439

   SIMPLE:term  SIMPLE:mort_acc  SIMPLE:num_tl_90g_dpd_24m  \
0     0.530784         0.574285                   0.497263
1     0.577225         0.583780                   0.511478
2     0.569453         0.557051                   0.500788
3     0.576510         0.539040                   0.511350
4     0.580011         0.586629                   0.504020
5     0.580407         0.588922                   0.514060
6     0.595200         0.579932                   0.495517
7     0.589895         0.499613                   0.502483
8     0.546206         0.539451                   0.484154
9     0.568945         0.563044                   0.494304

   SIMPLE:install_income  ALL_NUMERICAL
0               0.573958       0.618991
1               0.645416       0.678013
2               0.583904       0.634332
```

```
3               0.615912        0.694704
4               0.627962        0.674602
5               0.587645        0.678554
6               0.641287        0.679119
7               0.630371        0.655183
8               0.595181        0.618592
9               0.651754        0.670130
```

[34]:
```
plt.figure(figsize= (10,5))
plot_boxplot_ordered(model_perf)
plt.xlabel("Predictive Model with a Single Predictive Feature")
plt.ylabel("AUC")
```

[34]:
```
Text(0, 0.5, 'AUC')
```



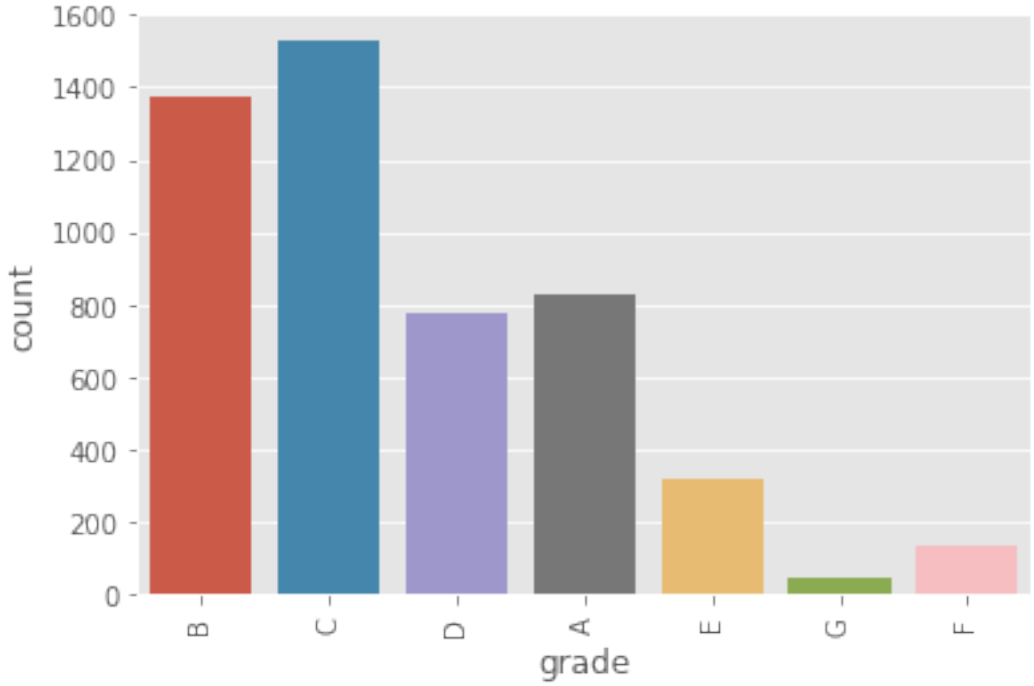
We see that the combined model does perform better than the best single-feature model. Thus, we will move forward with it for the rest of this case. Note, however, that best practice would entail iteratively adding features to the best single-feature model until we reach a point where there is no significant improvement, as opposed to throwing all the features in at once. We advise you to take this more cautious approach when building your own models.

1.5 Incorporating categorical variables (25 mts)

The grade of a loan (i.e. the LC-assigned loan grade feature) has not been used so far. The following is the distribution of the categorical grade feature:

[35]:
```
Data.emp_length.value_counts()
sns.countplot(x='grade', data = Data)
plt.xticks(rotation=90)
```

[35]:
```
(array([0, 1, 2, 3, 4, 5, 6]), <a list of 7 Text xticklabel objects>)
```



1.5.1 Exercise 8: (5 mts)

8.1 Use `pandas.get_dummies()` to transform this into its one-hot encoded version.

Answer. One possible solution is shown below:

[36]:
```
#use a one-hot-encoding approach for incorporating the "grade" categorical␣
↪variable
grade_categ = pd.get_dummies(Data['grade'], prefix = "grade", drop_first=True)
grade_categ.head()
```

[36]:

	grade_B	grade_C	grade_D	grade_E	grade_F	grade_G
3394	1	0	0	0	0	0

3206	0	1	0	0	0	0
4462	0	1	0	0	0	0
1093	0	0	1	0	0	0
1555	0	0	0	0	0	0

8.2 Add this feature to the all-numerical model from earlier and investigate whether this leads to a significant increase in predictive accuracy.

Answer. One possible solution is shown below:

[37]:
```
X_grade = pd.concat([X_full,grade_categ],axis=1)
X_grade.head()
```

[37]:

	annual_inc	avg_cur_bal	chargeoff_within_12_mths	delinq_2yrs	dti	\
3394	5.161368	3.611511	0.0	0.0	6.21	
3206	5.000000	3.700444	0.0	2.0	30.07	
4462	4.698970	3.253580	0.0	0.0	16.19	
1093	4.875061	4.068297	0.0	1.0	37.70	
1555	4.929419	4.352665	0.0	0.0	24.87	

	emp_length	inq_last_12m	loan_amnt	num_actv_bc_tl	\
3394	10	4.0	4.103804	1.0	
3206	1	0.0	3.885078	4.0	
4462	6	3.0	4.158362	6.0	
1093	6	4.0	4.176091	3.0	
1555	4	4.0	4.000000	1.0	

	pub_rec_bankruptcies	term	mort_acc	num_tl_90g_dpd_24m	\
3394	2.0	36	0.0	0.0	
3206	0.0	36	0.0	0.0	
4462	0.0	36	0.0	0.0	
1093	0.0	36	0.0	0.0	
1555	0.0	36	2.0	0.0	

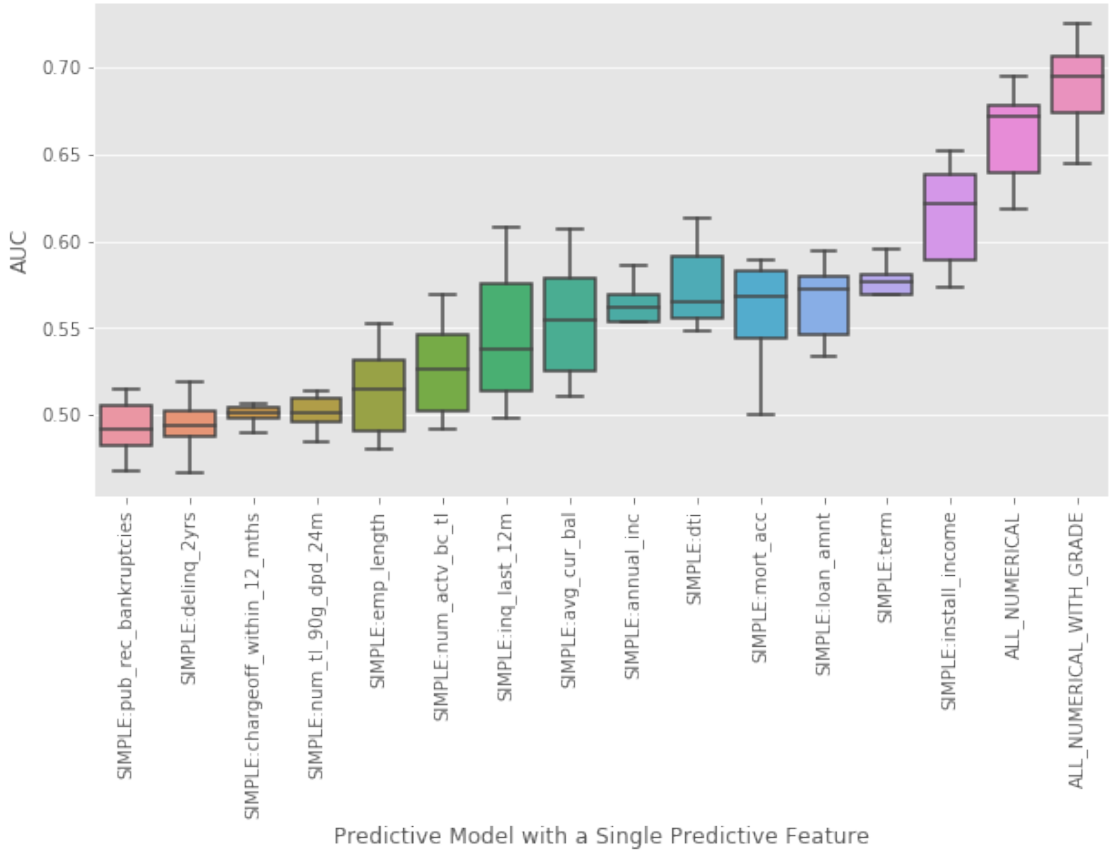
	install_income	grade_B	grade_C	grade_D	grade_E	grade_F	grade_G
3394	0.033791	1	0	0	0	0	0
3206	0.030587	0	1	0	0	0	0
4462	0.116098	0	1	0	0	0	0
1093	0.088619	0	0	1	0	0	0
1555	0.042515	0	0	0	0	0	0

[38]:
```
auc_list = cross_validation_AUC(X_grade, y_full)
model_perf["ALL_NUMERICAL_WITH_GRADE"] = auc_list
```

[39]:
```
plt.figure(figsize= (10,5))
plot_boxplot_ordered(model_perf)
plt.xlabel("Predictive Model with a Single Predictive Feature")
```

```
plt.ylabel("AUC")
```

[39]: `Text(0, 0.5, 'AUC')`



The difference appears significant as the boxplot for the updated model is almost completely non-overlapping with that of the previous model.

1.5.2 Exercise 9: (15 mts)

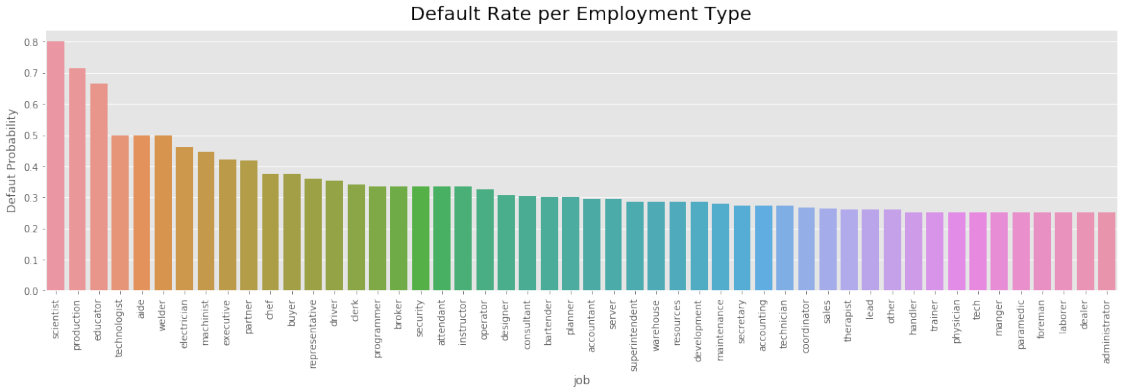
Investigate whether the categorical variable `job` brings any predictive value when added to the current best model. Again, you may want to use a one-hot encoding scheme.

Answer. One possible solution is shown below:

[40]:
```
plt.figure(figsize= (20,5))

df_job_default = Data[["loan_default", "job"]].groupby("job").mean()
df_job_default = df_job_default.sort_values(by="loan_default",axis=0,␣
 ↪ascending=False)
sns.barplot(x=df_job_default.index[:50],
```

```
            y=df_job_default["loan_default"][:50].values,
            orient="v")
plt.xticks(rotation=90)

plt.ylabel("Defaut Probability")
plt.title("Default Rate per Employment Type", fontsize=20,␣
 ↪verticalalignment='bottom')
```

[40]:
```
Text(0.5, 1.0, 'Default Rate per Employment Type')
```



[41]:
```
#use a one-hot-encoding approach for incorporating the "purpose" categorical␣
 ↪variable
job_categ = pd.get_dummies(Data['job'], prefix = "job", drop_first=True)
```

[42]:
```
X_grade_job = pd.concat([X_grade,job_categ],axis=1)
X_grade_job.head()
```

[42]:

	annual_inc	avg_cur_bal	chargeoff_within_12_mths	delinq_2yrs	dti	\
3394	5.161368	3.611511	0.0	0.0	6.21	
3206	5.000000	3.700444	0.0	2.0	30.07	
4462	4.698970	3.253580	0.0	0.0	16.19	
1093	4.875061	4.068297	0.0	1.0	37.70	
1555	4.929419	4.352665	0.0	0.0	24.87	

	emp_length	inq_last_12m	loan_amnt	num_actv_bc_tl	\
3394	10	4.0	4.103804	1.0	
3206	1	0.0	3.885078	4.0	
4462	6	3.0	4.158362	6.0	
1093	6	4.0	4.176091	3.0	
1555	4	4.0	4.000000	1.0	

	pub_rec_bankruptcies	…	job_technician	job_technologist	job_teller	\
3394	2.0	…	0	0	0	

```
3206                   0.0  …                0                 0           0
4462                   0.0  …                0                 0           0
1093                   0.0  …                0                 0           0
1555                   0.0  …                0                 0           0

      job_therapist  job_trainer  job_underwriter  job_vp  job_warehouse  \
3394              0            0                0       0              0
3206              0            0                0       0              0
4462              0            0                0       0              0
1093              0            0                0       0              0
1555              0            0                0       0              0

      job_welder  job_worker
3394           0           0
3206           0           0
4462           0           0
1093           0           0
1555           0           0

[5 rows x 138 columns]
```

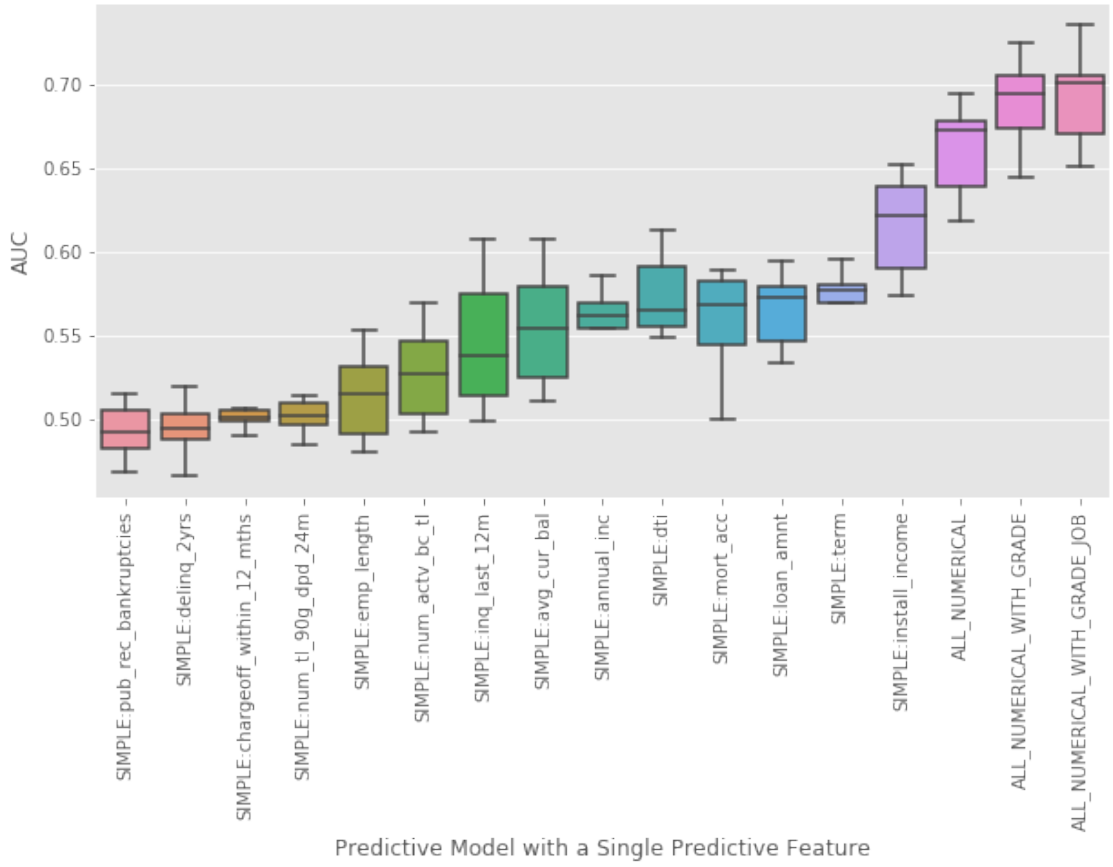
[43]:
```
X_grade_job.keys()
# --> 138 features
```

[43]:
```
Index(['annual_inc', 'avg_cur_bal', 'chargeoff_within_12_mths', 'delinq_2yrs',
       'dti', 'emp_length', 'inq_last_12m', 'loan_amnt', 'num_actv_bc_tl',
       'pub_rec_bankruptcies',
       …
       'job_technician', 'job_technologist', 'job_teller', 'job_therapist',
       'job_trainer', 'job_underwriter', 'job_vp', 'job_warehouse',
       'job_welder', 'job_worker'],
      dtype='object', length=138)
```

[44]:
```
auc_list = cross_validation_AUC(X_grade_job, y_full)
model_perf["ALL_NUMERICAL_WITH_GRADE_JOB"] = auc_list
```

[45]:
```
plt.figure(figsize= (10,5))
plot_boxplot_ordered(model_perf)
plt.xlabel("Predictive Model with a Single Predictive Feature")
plt.ylabel("AUC")
```

[45]:
```
Text(0, 0.5, 'AUC')
```



We can see that the boxplots overlap significantly, so there is no discernable benefit. We can repeat this process with other categorical variables to iteratively build the simplest possible model.

1.6 Conclusions (5 mts)

In this case, we first explored the loan dataset and found the single-variable associations between the available features and the default rate. We also discovered which features required transformations (e.g. log transform).

Once we started building models, we started with very simple logistic regressions approaches – these baseline models were useful for quickly evaluating the predictive power of each individual variable. Next, we employed cross-validation approaches for building more complex models, often exploiting the interactions between the different features. Since the loan dataset contains a large number of covariates, using cross-validation was revealed to be crucial for avoiding overfitting, choosing the correct number of features and ultimately choosing an appropriate model that balanced complexity with accuracy.

1.7 Takeaways (5 mts)

Cross-validation is a robust and flexible technique for evaluating the predictive performance of statistical models. It is especially useful in big data settings where the number of features is large compared to the number of observations. When used appropriately, cross-validation is a powerful method for choosing a model with the correct complexity and best predictive performance.