Deep Learning Project - Predicting Loan Default

August 21, 2020

1 Deep Learning Project - Predicting Loan Default

The Data:
For this project I will be using a subset of the LendingClub DataSet obtained from Kaggle: https://www.kaggle.com/wordsforthewise/lending-club. The data from this source has already been preprocessed for the use of this project.
Project Goal:
Given historical data on loans given out with information on whether or not the borrower defaulted, I will build a model that can predict wether or nor a borrower will pay back their loan. This way in the future when we get a new potential customer we can assess whether or not they are likely to pay back the loan. For this classification task I am going to use **keras** library and create a deep learning predictive model.
The "loan_status" column contains our label.

1.1 Importing Libraries and Loading Data

[3]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import re
```

[5]:
```
# Loading Lending Club Data Set
path = 'D:\Professional\_My Projects\Deep Learning Project - Predicting Loan␣
↪Default'
path_to_file = (path + '\lending_club_loan.csv')
df = pd.read_csv(path_to_file)
```

1.2 ## Exploratory Data Analysis

[38]:
```
df.info()
```

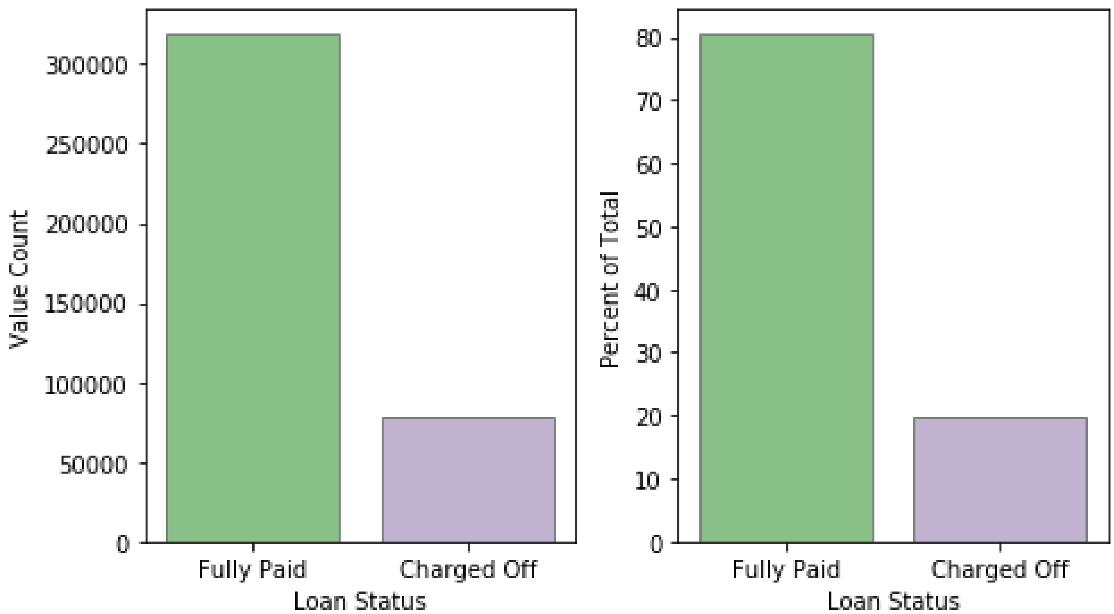
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 396030 entries, 0 to 396029
Data columns (total 27 columns):
loan_amnt               396030 non-null float64
term                    396030 non-null object
int_rate                396030 non-null float64
installment             396030 non-null float64
grade                   396030 non-null object
sub_grade               396030 non-null object
emp_title               373103 non-null object
emp_length              377729 non-null object
home_ownership          396030 non-null object
annual_inc              396030 non-null float64
verification_status     396030 non-null object
issue_d                 396030 non-null object
loan_status             396030 non-null object
purpose                 396030 non-null object
title                   394275 non-null object
dti                     396030 non-null float64
earliest_cr_line        396030 non-null object
open_acc                396030 non-null float64
pub_rec                 396030 non-null float64
revol_bal               396030 non-null float64
revol_util              395754 non-null float64
total_acc               396030 non-null float64
initial_list_status     396030 non-null object
application_type        396030 non-null object
mort_acc                358235 non-null float64
pub_rec_bankruptcies    395495 non-null float64
address                 396030 non-null object
dtypes: float64(12), object(15)
memory usage: 81.6+ MB
```

We have a good number of both numeric and text columns. We can begin with visualizing some numeric columns to get a better understanding of our data.

First, let's take a look at the distribution of our target column **loan_status**.

[32]:
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize = (7,4))
sns.countplot(data=df, x = 'loan_status', palette = 'Accent', edgecolor =␣
↪'gray', ax = ax1)
sns.barplot(data=df, x='loan_status', y='loan_amnt', palette = 'Accent',␣
↪edgecolor = 'gray',
            estimator=lambda x: len(x)/len(df)*100, ax=ax2)
plt.tight_layout()
ax1.set(ylabel='Value Count', xlabel='Loan Status')
ax2.set(ylabel='Percent of Total', xlabel='Loan Status')
```

[32]:
```
[Text(252.4295454545454, 0.5, 'Percent of Total'),
 Text(0.5, 15.0, 'Loan Status')]
```

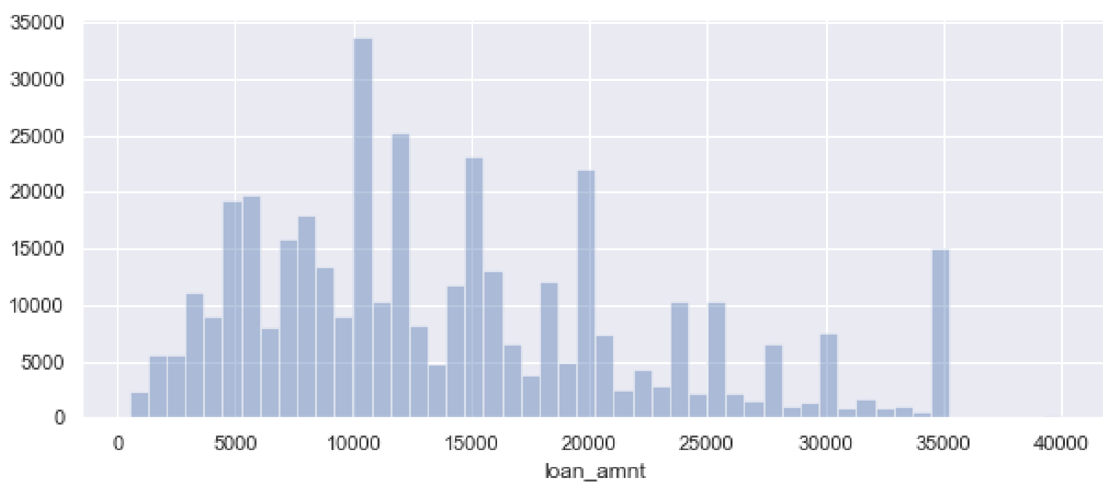


We can see that our target variable is significantly unbalanced, with around 80% of all values being 'Fully Paid'. It means that out future predictive model will be very accurate by default, because if we simply predict every loan to be Fully Paid we get 80% of accuracy. **Therefore, 80% accuracy is the lower limit of model performance.**

Next, we plot the distribution of Loan Ammount data column.

[34]:
```
plt.figure(figsize = (10,4))
sns.distplot(df['loan_amnt'], kde = False)
```

[34]: `<matplotlib.axes._subplots.AxesSubplot at 0x1bef13f4048>`

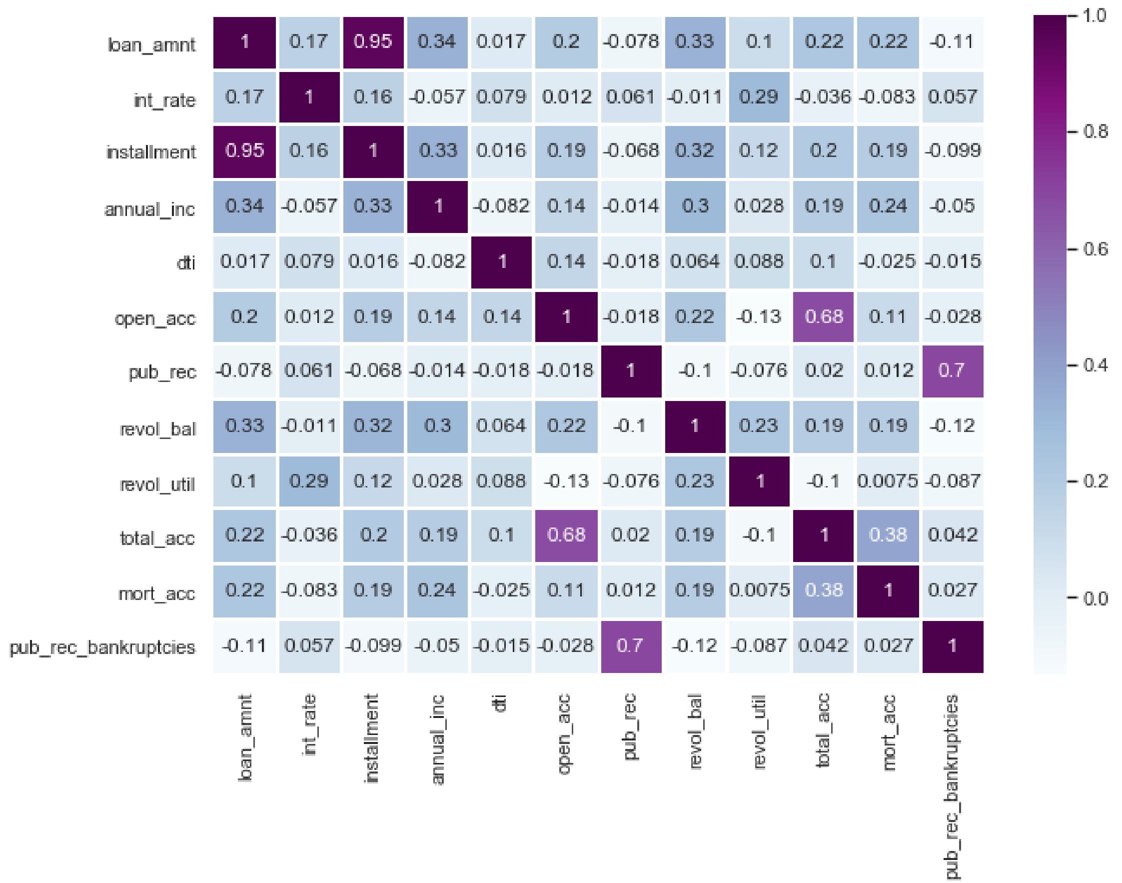


We see that there are popular, more frequent amounts of loans, such as, 5k, 10k, 15k, 20k, 25k and 35k. This makes sense, because more often people get a round amount of money.

Now let's explore correlation between the continuous feature variables. A better way to look at it is using a seaborn heatmap.

[45]:
```
plt.figure(figsize = (10,7))
sns.heatmap(df.corr(), linecolor = 'white', linewidth = 1, cmap = 'BuPu', annot = df.corr())
```

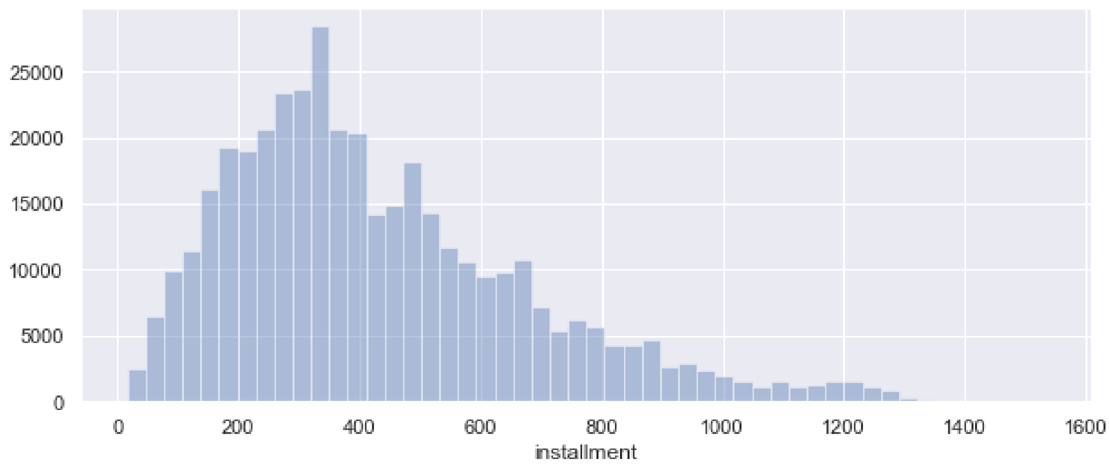
[45]: <matplotlib.axes._subplots.AxesSubplot at 0x1bef2638908>



We can see an almost perfect correlation between the "installment" and "loan_amnt" features. Let's explore these features further.

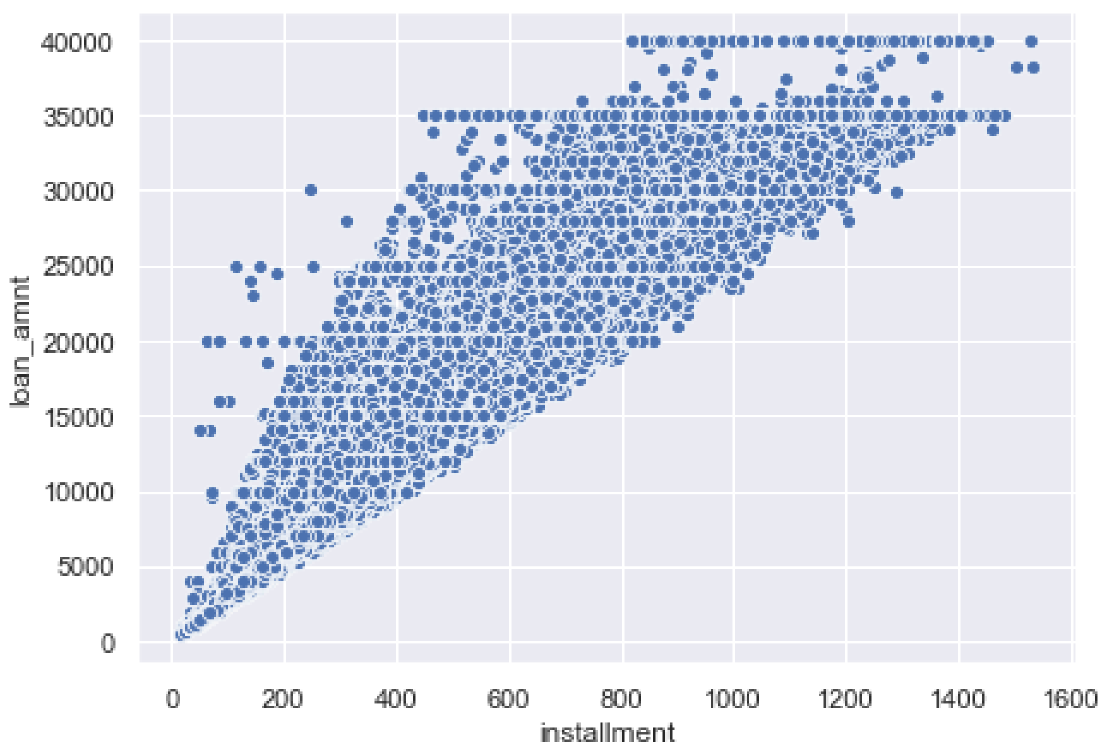
[48]:
```
plt.figure(figsize=(10,4))
sns.distplot(df['installment'], kde = False)
```

[48]: <matplotlib.axes._subplots.AxesSubplot at 0x1bef27c72b0>



[49]:
```
plt.figure(figsize=(7,5))
sns.scatterplot(data=df,x='installment',y='loan_amnt')
```

[49]: <matplotlib.axes._subplots.AxesSubplot at 0x1bef2af4908>



The above scatterplot show the direct linear relationship between the two features. It makes sense tha the installment amount is calculated as a functuon of the loan amount.

Let's take a look at the summary statistics for the loan amount, grouped by the loan_status.

[50]:
```
df.groupby('loan_status').describe().loc[:,'loan_amnt']
```

[50]:
```
                 count          mean          std     min     25%      50%  \
loan_status
Charged Off    77673.0  15126.300967  8505.090557  1000.0  8525.0  14000.0
Fully Paid    318357.0  13866.878771  8302.319699   500.0  7500.0  12000.0

                 75%      max
loan_status
Charged Off  20000.0  40000.0
Fully Paid   19225.0  40000.0
```

The average loan amount for defaulted cases is lower, which makes sense - the larget loan is more difficult to pay back.

Now let's explore the Grade and SubGrade columns that LendingClub attributes to the loans. We create a list of ordered grades to apply to our visuals and make them more meaningful.

[52]:
```
grade_reordered = sorted(df['grade'].unique())
print(grade_reordered)
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

[53]:
```
sub_grade_reordered = sorted(df['sub_grade'].unique())
print(sub_grade_reordered)
```

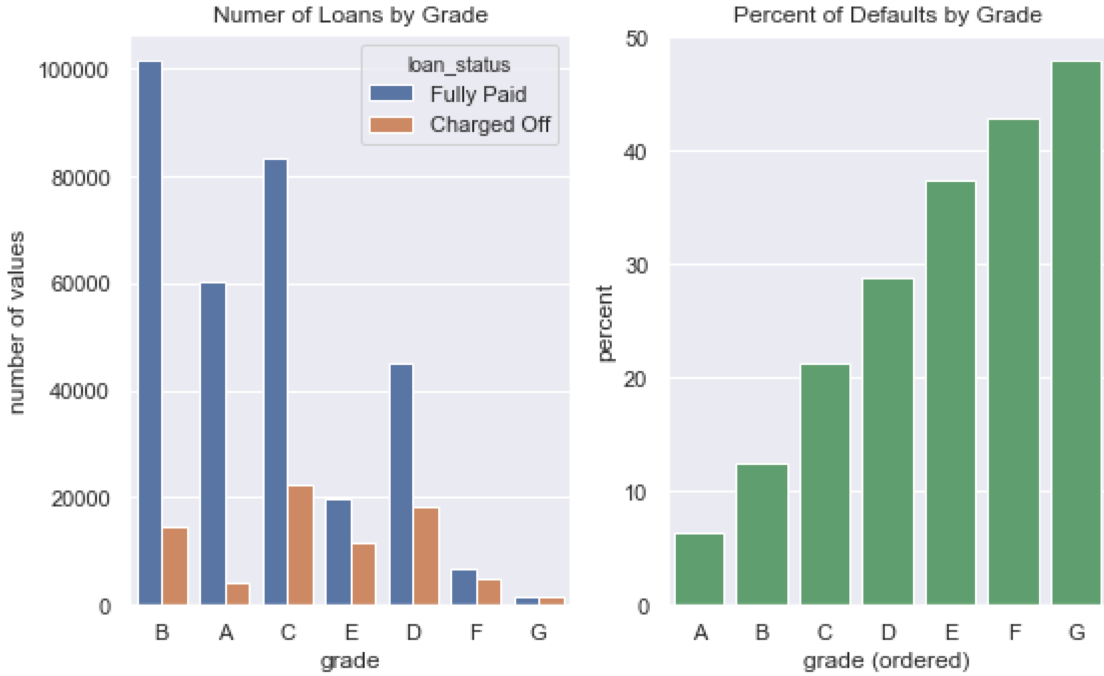
```
['A1', 'A2', 'A3', 'A4', 'A5', 'B1', 'B2', 'B3', 'B4', 'B5', 'C1', 'C2', 'C3',
'C4', 'C5', 'D1', 'D2', 'D3', 'D4', 'D5', 'E1', 'E2', 'E3', 'E4', 'E5', 'F1',
'F2', 'F3', 'F4', 'F5', 'G1', 'G2', 'G3', 'G4', 'G5']
```

[657]:
```
grade_reordered = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

[54]:
```
grades_paid = df[df['loan_status']=='Fully Paid']['loan_status'].
 ↪groupby(df['grade']).count()
grades_notpaid = df[df['loan_status']=='Charged Off']['loan_status'].
 ↪groupby(df['grade']).count()
grades_perc = grades_notpaid / (grades_notpaid + grades_paid) * 100
```

[66]:
```
# CODE HERE
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize = (8,5))
sns.countplot(data=df, x='grade', hue='loan_status', edgecolor = 'white', ax =␣
 ↪ax1)
sns.barplot(x=grades_perc.index, y=grades_perc.values, edgecolor = 'white', ax␣
 ↪= ax2, color = 'g')
ax2.set(title='Percent of Defaults by Grade', ylabel = 'percent', xlabel =␣
 ↪'grade (ordered)')
ax1.set(title='Numer of Loans by Grade', ylabel='number of values')
```

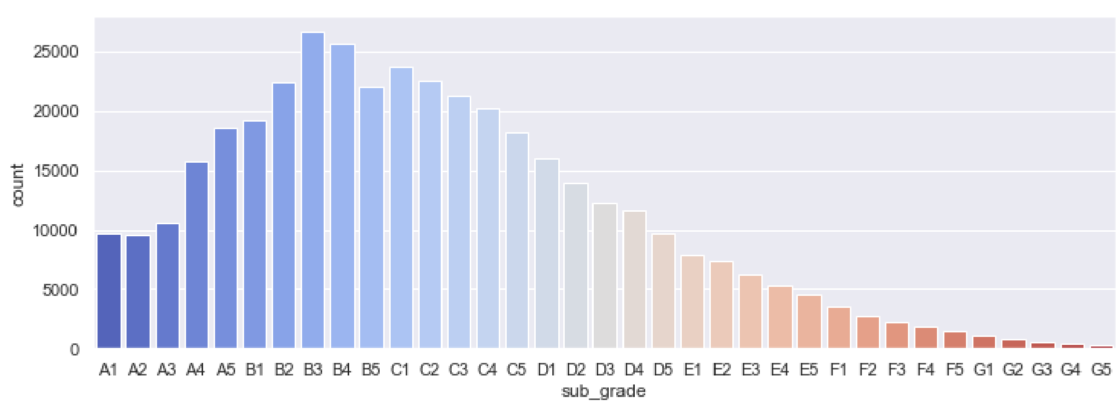
```
plt.tight_layout()
```



We see a very clear relationship - as the loan grade goes down from A to G, the chance of default for this loan is increasing.

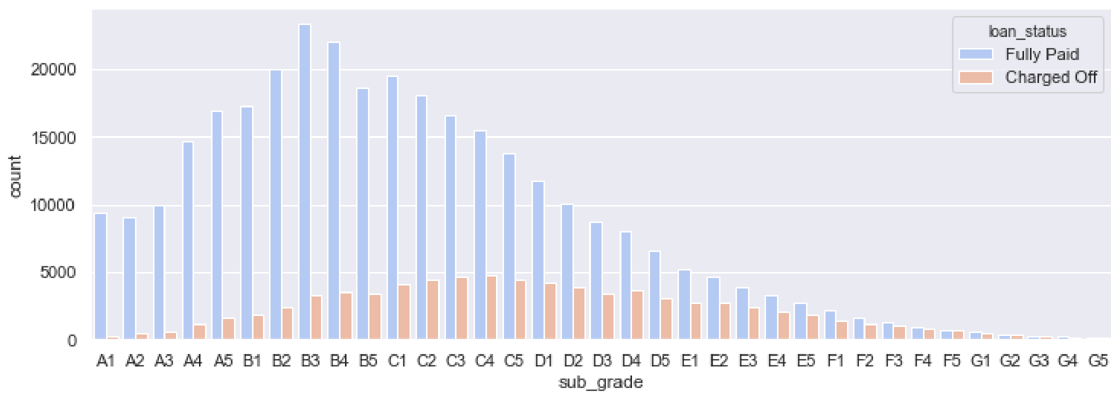
[70]:
```
plt.figure(figsize=(12,4))
sns.countplot(data=df,␣
↪x='sub_grade',order=sub_grade_reordered,palette='coolwarm')
```

[70]: <matplotlib.axes._subplots.AxesSubplot at 0x1be86738358>



[72]:
```
plt.figure(figsize=(12,4))
sns.countplot(data=df,␣
↪x='sub_grade',order=sub_grade_reordered,palette='coolwarm',hue='loan_status'␣
↪)
```

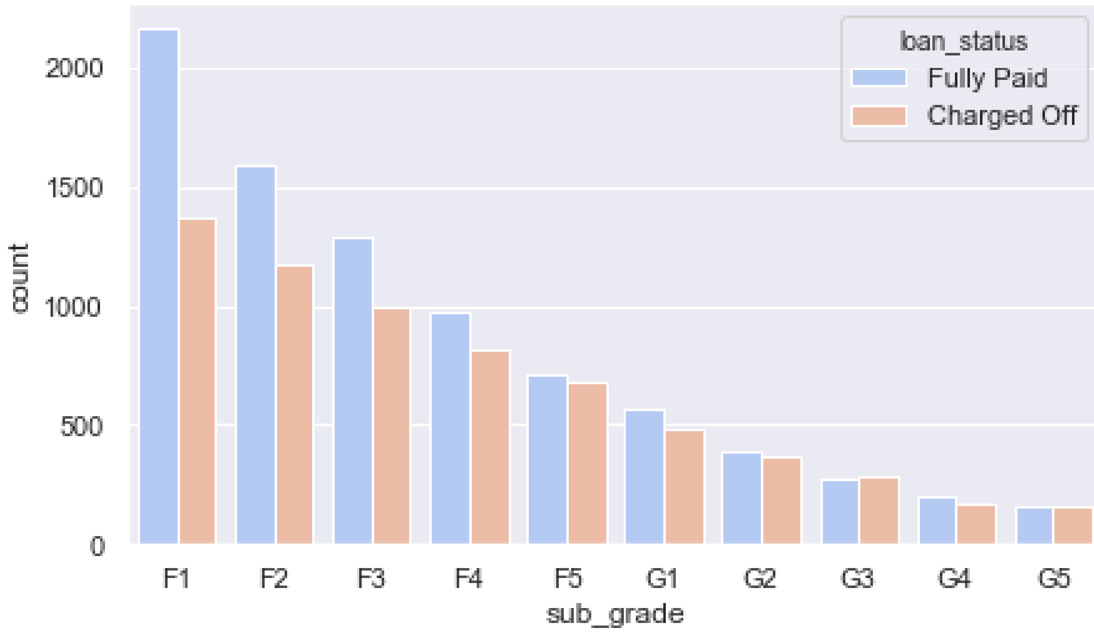
[72]: <matplotlib.axes._subplots.AxesSubplot at 0x1be86d397f0>



It looks like F and G subgrades don't get paid back that often. Let's isolate those and recreate the countplot just for those subgrades.

[75]:
```
# isolate Fs and Gs
df_1 = df[(df['sub_grade'].str.contains("F")) | (df['sub_grade'].str.
↪contains("G"))]
# create order list
l2=sorted(df_1['sub_grade'].unique())
# plot
plt.figure(figsize=(7,4))
sns.countplot(data=df_1,␣
↪x='sub_grade',palette='coolwarm',hue='loan_status',order=l2)
```

[75]: <matplotlib.axes._subplots.AxesSubplot at 0x1be87120860>



Now let's create a new column called 'loan_repaid' which will contain a 1 if the loan status was "Fully Paid" and a 0 if it was "Charged Off".

[98]:
```
#most efficient way to do it
df['loan_repaid'] = df['loan_status'].map({'Fully Paid':1, 'Charged Off':0})
```

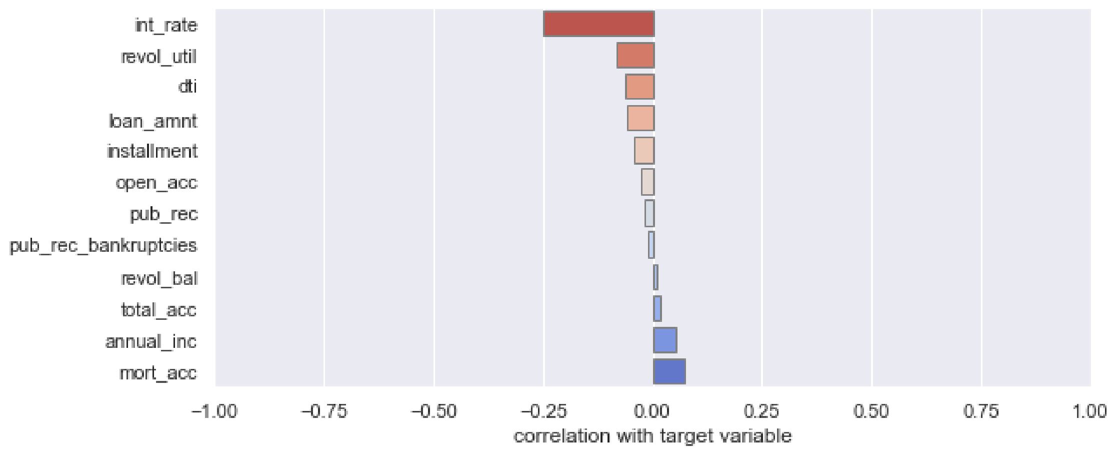
[101]:
```
df.loc[:,['loan_repaid','loan_status']].head()
```

[101]:
```
   loan_repaid  loan_status
0            1   Fully Paid
1            1   Fully Paid
2            1   Fully Paid
3            1   Fully Paid
4            0  Charged Off
```

Now we can check the correlation of numeric features with a newly created "loan_repaid" variable.

[106]:
```
my_corr = df.corr().loc[:,'loan_repaid']
my_corr = my_corr.sort_values(ascending=True)
plt.figure(figsize=(9,4))
sns.barplot(y = my_corr[:-1].index, x = my_corr[:-1], orient = 'h', palette =␣
↪'coolwarm_r', edgecolor = 'gray')
plt.xlim(-1, 1)
plt.xlabel('correlation with target variable')
```

[106]:
```
Text(0.5, 0, 'correlation with target variable')
```



We can notice that our numeric variables have very week correlation with the target variable. The only only feature that has a mild correlation with "loan status" is "interest rate".

Let's use **Predictive Power Score** to understand how effective our features at predicting the loan statu.

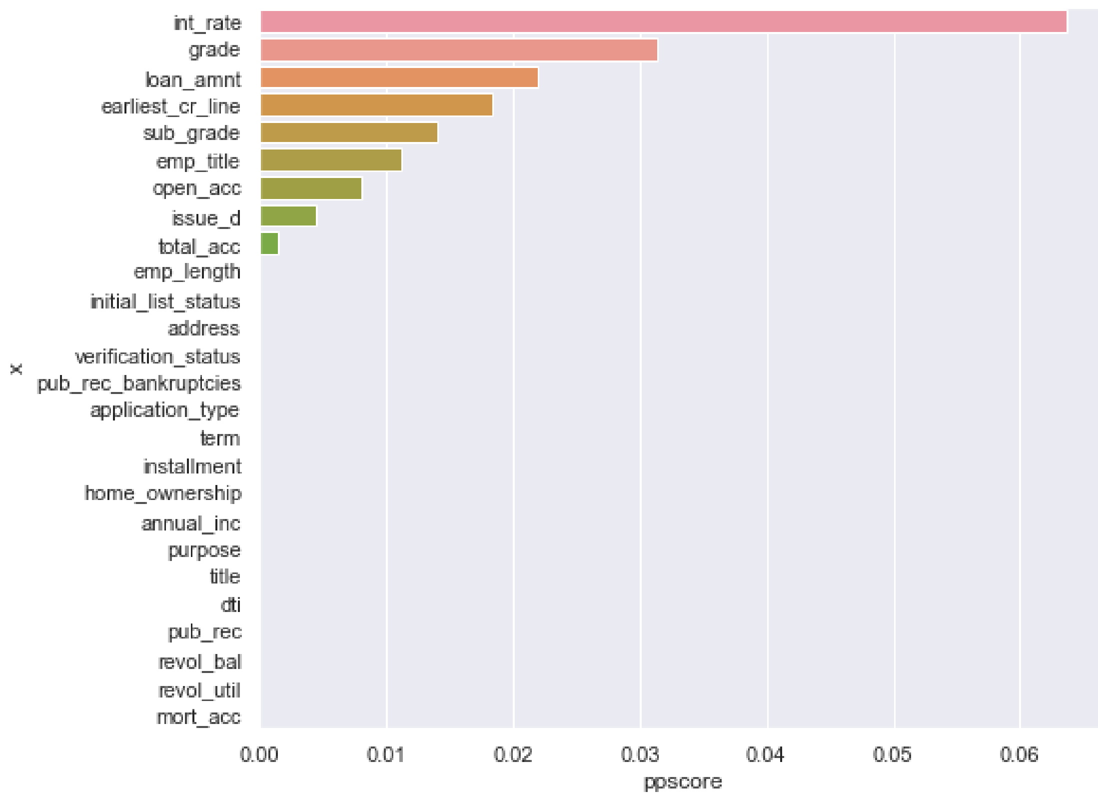
[107]:
```
import ppscore as pps
```

[128]:
```
# There are multiple warnings about ill-defined F1 score which do not mess up␣
↪the result in any way, so let's ignore them
import warnings
with warnings.catch_warnings():
    warnings.simplefilter('ignore')
    predictors_df = pps.predictors(df, y="loan_status")
```

[129]:
```
predictors_df = predictors_df.iloc[1:]
```

[127]:
```
plt.figure(figsize=(8,7))
sns.barplot(data=predictors_df, y="x", x="ppscore", orient = 'h')
```

[127]: <matplotlib.axes._subplots.AxesSubplot at 0x1be8d53c9b0>



As we can see, most features have PPS of zero, which means they can't predict the target better than naive model. Those few variables that have nonzero PPS are still very weak. **None of the features stand out as a good predictor judging by Predictive Power Score.**

1.3 Data PreProcessing

During this part of the project we are going to: 1. Remove or fill any missing data. 2. Remove unnecessary or repetitive features. 3. Convert categorical string features to dummy variables.

[103]:
```
# 1. Remove or fill any missing data
# what percent of my data is missing per column ?
round(df.isna().sum()[df.isna().sum() > 0] / len(df) * 100,1)
```

[103]:
```
emp_title               5.8
emp_length              4.6
title                   0.4
revol_util              0.1
mort_acc                9.5
pub_rec_bankruptcies    0.1
dtype: float64
```

[104]:
```
# how much of the data do I lose if I just drop all rows with missing data
round((1 - len(df.dropna())/len(df)) * 100,1)
```

[104]: 15.2

If we just drop all missing values we will lose 15% of the data. That is too much data to be lost. We need to come up with a better way to deal with missing values.

1.3.1 Missing Data

Let's explore our missing data columns. We use a variety of factors to decide whether or not they would be useful, to see if we should keep, discard, or fill in the missing data.

First let's examine emp_title and emp_length to see whether it will be okay to drop them. Let's see how many unique values are there.

[133]:
```
df['emp_title'].nunique()
```

[133]: 173105

[134]:
```
df['emp_title'].value_counts().head()
```

[134]:
```
Teacher              4389
Manager              4250
Registered Nurse     1856
RN                   1846
Supervisor           1830
Name: emp_title, dtype: int64
```

Realistically there are too many unique job titles to try to convert this to a dummy variable feature. Let's remove that emp_title column.

[135]:
```
del df['emp_title']
```

Now let's examine the employment length feature - "emp_length".

[138]:
```
df['emp_length'].unique()
```

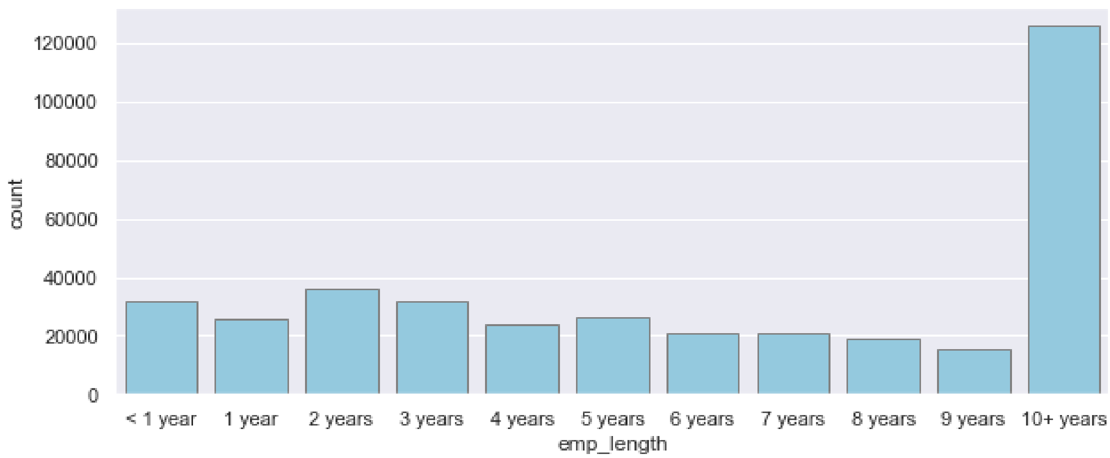
[138]:
```
array(['10+ years', '4 years', '< 1 year', '6 years', '9 years',
       '2 years', '3 years', '8 years', '7 years', '5 years', '1 year',
       nan], dtype=object)
```

It would be better to order these values, thus we can visualize it better.

[139]:
```
order_list = ['< 1 year', '1 year', '2 years', '3 years', '4 years', '5 years',
              '6 years', '7 years', '8 years', '9 years', '10+ years']
```

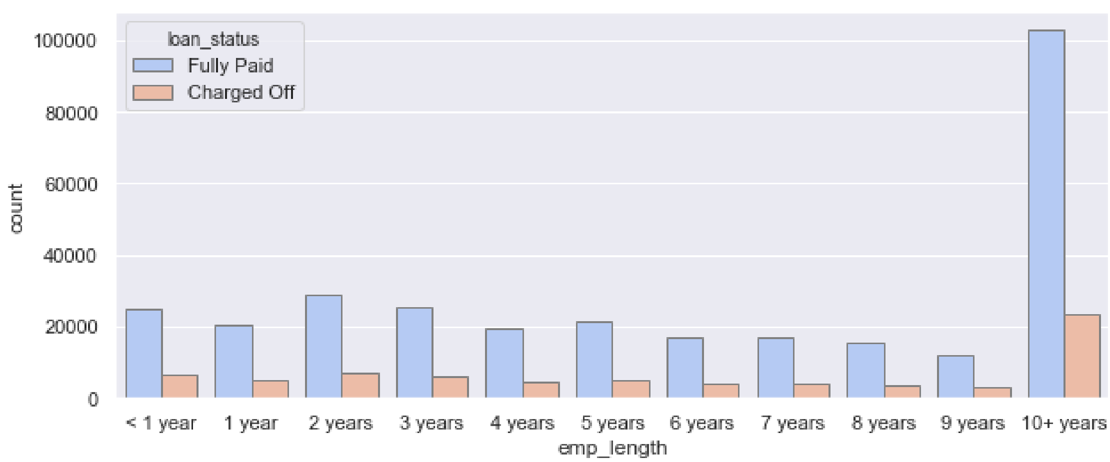
[141]:
```
plt.figure(figsize=(10,4))
sns.countplot(data=df, x = 'emp_length', order = order_list, color = 'skyblue',␣
 ↪edgecolor = 'gray')
```

[141]: <matplotlib.axes._subplots.AxesSubplot at 0x1be8d624b00>



[143]:
```
plt.figure(figsize=(10,4))
sns.countplot(data=df, x = 'emp_length', hue = 'loan_status',
              order = order_list, palette = 'coolwarm', edgecolor = 'gray')
```

[143]: <matplotlib.axes._subplots.AxesSubplot at 0x1be8d748dd8>



This still doesn't really inform us if there is a strong relationship between employment length and being charged off, what we want is the percentage of charge offs per category. Essentially informing us what percent of people per employment category didn't pay back their loan.

[144]:
```
emp_co = df[df['loan_status']=='Charged Off'].groupby('emp_length').
 ↪count()['loan_status']
emp_fp = df[df['loan_status']=='Fully Paid'].groupby('emp_length').
 ↪count()['loan_status']
```

[145]:
```
perc_fp = emp_co/(emp_fp+emp_co)*100
```

[147]:
```
plt.figure(figsize=(9,4))
sns.barplot(x = perc_fp.index, y = perc_fp.values,
            color = 'skyblue', edgecolor = 'gray', order = order_list)
plt.ylim(15,22)
plt.xlabel('Employment Length')
plt.ylabel('Percent of Total Loans')
plt.title('Percent of Repaid Loans across Employment Length Groups', fontsize =
 ↪15)
```

[147]:
```
Text(0.5, 1.0, 'Percent of Repaid Loans across Employment Length Groups')
```

Charge off rates are extremely similar across all employment lengths, so it's not a good predictor. We can drop the emp_length column.

[148]:
```
del df['emp_length']
```

Let's see what feature columns still have missing data.

[149]:
```
df.isna().sum()[df.isna().sum() > 0]
```

[149]:
```
title                    1755
revol_util                276
mort_acc                37795
pub_rec_bankruptcies      535
dtype: int64
```

If we take a closer look at "purpose" and "title" features we can see that they have the same information.

[150]:
```
df.loc[:,['purpose','title']].head()
```

[150]:
```
              purpose                 title
0            vacation              Vacation
1  debt_consolidation    Debt consolidation
```

```
2          credit_card  Credit card refinancing
3          credit_card  Credit card refinancing
4          credit_card    Credit Card Refinance
```

The title column is simply a string subcategory/description of the purpose column, so we can drop the "title" column.

[151]:
```
df = df.drop('title', axis = 1)
```

Now let's explore the "mort_acc" feature.

[156]:
```
df['mort_acc'].value_counts().head(5)
```

[156]:
```
0.0    139779
1.0     60416
2.0     49949
3.0     38049
4.0     27887
Name: mort_acc, dtype: int64
```

Let's review the other columsn to see which most highly correlates to mort_acc.

[153]:
```
mort_corr = df.corr().loc[:,'mort_acc'].sort_values(ascending=True)
mort_corr[:-1]
```

[153]:
```
int_rate               -0.082583
dti                    -0.025439
revol_util              0.007514
pub_rec                 0.011552
pub_rec_bankruptcies    0.027239
loan_repaid             0.073111
open_acc                0.109205
installment             0.193694
revol_bal               0.194925
loan_amnt               0.222315
annual_inc              0.236320
total_acc               0.381072
Name: mort_acc, dtype: float64
```

Looks like the total_acc feature correlates with the mort_acc! Let's try the fillna() approach. We will group the dataframe by the total_acc and calculate the mean value for the mort_acc per total_acc entry.

[154]:
```
amg = df['mort_acc'].groupby(df['total_acc']).mean()
```

Let's fill in the missing mort_acc values based on their total_acc value. If the mort_acc is missing, then we will fill in that missing value with the mean value corresponding to its total_acc value from the Series we created above. This involves using an .apply() method with two columns.

[155]:
```
df['mort_acc'] = df.apply(
    lambda row: amg[amg.index == row['total_acc']].values[0] if np.
 ↪isnan(row['mort_acc']) else row['mort_acc'],
```

```
axis=1)
```

Only rwo columns with missing values are left:

[160]:
```
round(df.isna().sum()[df.isna().sum() > 0] / len(df) * 100,2)
```

[160]:
```
revol_util               0.07
pub_rec_bankruptcies     0.14
dtype: float64
```

revol_util and the pub_rec_bankruptcies have missing data points, but they account for less than 0.5% of the total data. We can remove the rows that are missing those values with dropna().

[161]:
```
df = df.dropna(subset = ['revol_util', 'pub_rec_bankruptcies'], axis = 0)
```

1.3.2 Categorical Variables and Dummy Variables

We're done working with the missing data! Now we just need to deal with the string values due to the categorical columns. Let's list all the columns that are currently non-numeric.

[162]:
```
df.select_dtypes(include=['object']).columns
```

[162]:
```
Index(['term', 'grade', 'sub_grade', 'home_ownership', 'verification_status',
       'issue_d', 'loan_status', 'purpose', 'earliest_cr_line',
       'initial_list_status', 'application_type', 'address'],
      dtype='object')
```

1. term feature

[165]:
```
df['term'].unique()
```

[165]:
```
array([' 36 months', ' 60 months'], dtype=object)
```

We can convert the term feature into either a 36 or 60 integer numeric data type.

[166]:
```
df['term'] = df['term'].apply(lambda x: int(re.findall(r'\d+', x)[0]))
```

2. grade feature

Grade is part of sub_grade, so let's just drop the grade feature.

[167]:
```
df = df.drop('grade', axis = 1)
```

As for the subgrade feature, we can convert it into dummy variables, then concatenate these new columns to the original dataframe.

[164]:
```
subgrade_dummies = pd.get_dummies(df['sub_grade'], drop_first = True)
df = pd.concat([df, subgrade_dummies], axis = 1)
df = df.drop('sub_grade', axis=1)
```

[168]:
```
df.select_dtypes(include=['object']).columns
```

[168]:
```
Index(['home_ownership', 'verification_status', 'issue_d', 'loan_status',
       'purpose', 'earliest_cr_line', 'initial_list_status',
```

```
         'application_type', 'address'],
        dtype='object')
```

3. verification_status, application_type,initial_list_status,purpose

Again, we convert these columns: ['verification_status', 'application_type','initial_list_status','purpose'] into dummy variables and concatenate them with the original dataframe.

[169]:
```
# 1. verification status
col_name = 'verification_status'
my_dummies = pd.get_dummies(df[col_name], drop_first = True)
df = pd.concat([df, my_dummies], axis = 1)
df = df.drop(col_name, axis=1)
```

[170]:
```
# 2. application_type
col_name = 'application_type'
my_dummies = pd.get_dummies(df[col_name], drop_first = True)
df = pd.concat([df, my_dummies], axis = 1)
df = df.drop(col_name, axis=1)
```

[171]:
```
# 3. initial_list_status
col_name = 'initial_list_status'
my_dummies = pd.get_dummies(df[col_name], drop_first = True)
df = pd.concat([df, my_dummies], axis = 1)
df = df.drop(col_name, axis=1)
```

[172]:
```
# 4. purpose
col_name = 'purpose'
my_dummies = pd.get_dummies(df[col_name], drop_first = True)
df = pd.concat([df, my_dummies], axis = 1)
df = df.drop(col_name, axis=1)
```

[174]:
```
df.select_dtypes(include=['object']).columns
```

[174]:
```
Index(['home_ownership', 'issue_d', 'loan_status', 'earliest_cr_line',
       'address'],
      dtype='object')
```

4. home_ownership

[175]:
```
df['home_ownership'].value_counts()
```

[175]:
```
MORTGAGE    198022
RENT        159395
OWN          37660
OTHER          110
NONE            29
ANY              3
```

```
Name: home_ownership, dtype: int64
```

One more time we convert these to dummy variables, but replace NONE and ANY with OTHER, so that we end up with just 4 categories, MORTGAGE, RENT, OWN, OTHER. Then we concatenate them with the original dataframe.

[176]:
```
df['home_ownership'] = df['home_ownership'].replace(to_replace = ['NONE',␣
↪'ANY'], value = 'OTHER')
```

[177]:
```
col_name = 'home_ownership'
my_dummies = pd.get_dummies(df[col_name], drop_first = True)
df = pd.concat([df, my_dummies], axis = 1)
df = df.drop(col_name, axis=1)
```

[178]:
```
df.select_dtypes(include=['object']).columns
```

[178]:
```
Index(['issue_d', 'loan_status', 'earliest_cr_line', 'address'], dtype='object')
```

5. address
First, let's explore some values from address column.

[182]:
```
df['address'].head()
```

[182]:
```
0       0174 Michelle Gateway\nMendozaberg, OK 22690
1    1076 Carney Fort Apt. 347\nLoganmouth, SD 05113
2    87025 Mark Dale Apt. 269\nNew Sabrina, WV 05113
3              823 Reid Ford\nDelacruzside, MA 00813
4               679 Luna Roads\nGreggshire, VA 11650
Name: address, dtype: object
```

Let's feature engineer a zip code column from the address in the data set. I will create a column called 'zip_code' that extracts the zip code from the address column.

[183]:
```
df['zip_code'] = df['address'].apply(lambda x: x[-5:])
df['zip_code'].nunique()
```

[183]:
```
10
```

The next step is to make this zip_code column into dummy variables using pandas. Concatenate the result and drop the original zip_code column along with dropping the address column.

[184]:
```
col_name = 'zip_code'
my_dummies = pd.get_dummies(df[col_name], drop_first = True)
df = pd.concat([df, my_dummies], axis = 1)
df = df.drop(col_name, axis=1)
```

[185]:
```
df = df.drop('address', axis = 1)
```

[186]:
```
df.select_dtypes(include=['object']).columns
```

[186]:
```
Index(['issue_d', 'loan_status', 'earliest_cr_line'], dtype='object')
```

6. issue_d

[188]:
```python
df['issue_d'].head()
```

[188]:
```
0    Jan-2015
1    Jan-2015
2    Jan-2015
3    Nov-2014
4    Apr-2013
Name: issue_d, dtype: object
```

This column is the date when the loan was issued. This would be data leakage, we wouldn't know beforehand whether or not a loan would be issued when using our model, so in theory we wouldn't have an issue_date, let's drop this feature.

[189]:
```python
del df['issue_d']
```

7. earliest_cr_line

This appears to be a historical time stamp feature. Let's extract the year from this feature using a .apply function, then convert it to a numeric feature. Finally, we drop the earliest_cr_line feature.

[190]:
```python
df['earliest_cr_year'] = df['earliest_cr_line'].apply(lambda x: int(re.
 ↪findall(r'\d+', x)[0]))
```

[191]:
```python
del df['earliest_cr_line']
```

[197]:
```python
df.select_dtypes(include=['object']).columns
```

[197]:
```
Index([], dtype='object')
```

1.4 Train Test Split and Data Scaling

TASK: Import train_test_split from sklearn.

[193]:
```python
from sklearn.model_selection import train_test_split
```

Let's drop the load_status column, since its a duplicate of the loan_repaid column. We'll use the loan_repaid column since its already in 0s and 1s.

[194]:
```python
del df['loan_status']
```

We set X and y variables to the .values of the features and label.

[244]:
```python
X = df.drop('loan_repaid', axis = 1)
y = df['loan_repaid']
```

Then we perform a train/test split with test_size=0.2.

[245]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=101)
```

We need to normalize the feature data X_train and X_test to use in deep learning algorythms. For this task we will use a MinMaxScaler.

[247]:
```python
from sklearn.preprocessing import MinMaxScaler
```

[248]:
```python
scaler = MinMaxScaler()
```

[249]:
```
X_train = scaler.fit_transform(X_train)
```

[250]:
```
X_test = scaler.transform(X_test)
```

1.5 Predictive Modelling

Let's import the necessary Keras functions.

[251]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, TensorBoard
```

We will build a Sequential Model and train it on our data.

When you create a deep learning model and decide on the number of neurons for the first layer, a good rule of thumb is to look at the number of features and use the same amount of neurons.

For each next layer we will decrease the number of neurons by half. In total we will have 4 layers.

We are also using Dropout Layers to randomly exclude some layers from the model during training and avoid overfitting.

[261]:
```
pwd
```

[261]:
```
'C:\\Users\\iliai'
```

[262]:
```
log_directory = 'logs\\fit'
```

[263]:
```
board = TensorBoard(log_dir=log_directory,histogram_freq=1,
    write_graph=True,
    write_images=True,
    update_freq='epoch',
    profile_batch=2,
    embeddings_freq=1)
```

[264]:
```
model = Sequential()

# Choose whatever number of layers/neurons you want.
model.add(Dense(78, activation = 'relu'))
model.add(Dropout(0.3)) #to prevent overfitting

model.add(Dense(39, activation = 'relu'))
model.add(Dropout(0.3))

model.add(Dense(19, activation = 'relu'))
model.add(Dropout(0.3))

model.add(Dense(1, activation = 'sigmoid'))

model.compile(loss = 'binary_crossentropy', optimizer = 'adam')
```

In order to avoid overfitting we are also adding an early stop parameter. It will detect the moment when the model error goes up on validation data and stop training. The 'patience' parameter means that we will continue training model after the stop point due to the noise.

[265]:
```
# we are trying to minimize our validation loss
# and we will wait 10 epochs even after we detected stopping point because of␣
↪noise
# verbose = 1 gives some report data
early_stop = EarlyStopping(monitor='val_loss', mode='min',
                           verbose = 1, patience = 25)
```

It's time to fit the model.

[266]:
```
model.fit(x=X_train, y=y_train, epochs = 500, batch_size = 256,
          validation_data = (X_test, y_test),
          callbacks = [early_stop,board])
```

```
Epoch 1/500
   2/1236 [...] - ETA: 5:35 - loss:
0.7417WARNING:tensorflow:Callbacks method `on_train_batch_begin` is slow
compared to the batch time (batch time: 0.0000s vs `on_train_batch_begin` time:
0.0080s). Check your callbacks.
WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to the
batch time (batch time: 0.0000s vs `on_train_batch_end` time: 0.5354s). Check
your callbacks.
1236/1236 [==============================] - 4s 3ms/step - loss: 0.3108 -
val_loss: 0.2656
Epoch 2/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2678 -
val_loss: 0.2640
Epoch 3/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2646 -
val_loss: 0.2627
Epoch 4/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2631 -
val_loss: 0.2625
Epoch 5/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2621 -
val_loss: 0.2621
Epoch 6/500
1236/1236 [==============================] - 3s 2ms/step - loss: 0.2615 -
val_loss: 0.2623
Epoch 7/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2610 -
val_loss: 0.2622
Epoch 8/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2608 -
val_loss: 0.2619
Epoch 9/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2602 -
val_loss: 0.2623
Epoch 10/500
```

```
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2603 -
val_loss: 0.2625
Epoch 11/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2601 -
val_loss: 0.2616
Epoch 12/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2598 -
val_loss: 0.2616
Epoch 13/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2597 -
val_loss: 0.2614
Epoch 14/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2594 -
val_loss: 0.2614
Epoch 15/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2591 -
val_loss: 0.2612
Epoch 16/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2591 -
val_loss: 0.2613
Epoch 17/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2591 -
val_loss: 0.2612
Epoch 18/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2588 -
val_loss: 0.2620
Epoch 19/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2586 -
val_loss: 0.2613
Epoch 20/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2587 -
val_loss: 0.2613
Epoch 21/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2583 -
val_loss: 0.2612
Epoch 22/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2585 -
val_loss: 0.2615
Epoch 23/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2581 -
val_loss: 0.2616
Epoch 24/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2579 -
val_loss: 0.2612
Epoch 25/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2579 -
val_loss: 0.2611
Epoch 26/500
```

```
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2577 -
val_loss: 0.2634
Epoch 27/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2578 -
val_loss: 0.2612
Epoch 28/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2579 -
val_loss: 0.2621
Epoch 29/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2575 -
val_loss: 0.2614
Epoch 30/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2573 -
val_loss: 0.2619
Epoch 31/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2572 -
val_loss: 0.2612
Epoch 32/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2571 -
val_loss: 0.2613
Epoch 33/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2571 -
val_loss: 0.2610
Epoch 34/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2568 -
val_loss: 0.2614
Epoch 35/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2569 -
val_loss: 0.2612
Epoch 36/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2570 -
val_loss: 0.2611
Epoch 37/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2565 -
val_loss: 0.2614
Epoch 38/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2565 -
val_loss: 0.2611
Epoch 39/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2567 -
val_loss: 0.2613
Epoch 40/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2564 -
val_loss: 0.2614
Epoch 41/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2565 -
val_loss: 0.2613
Epoch 42/500
```

```
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2562 -
val_loss: 0.2612
Epoch 43/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2560 -
val_loss: 0.2620
Epoch 44/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2561 -
val_loss: 0.2613
Epoch 45/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2560 -
val_loss: 0.2610
Epoch 46/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2559 -
val_loss: 0.2615
Epoch 47/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2558 -
val_loss: 0.2610
Epoch 48/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2555 -
val_loss: 0.2609
Epoch 49/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2558 -
val_loss: 0.2610
Epoch 50/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2554 -
val_loss: 0.2609
Epoch 51/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2557 -
val_loss: 0.2609
Epoch 52/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2558 -
val_loss: 0.2617
Epoch 53/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2555 -
val_loss: 0.2607
Epoch 54/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2553 -
val_loss: 0.2609
Epoch 55/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2551 -
val_loss: 0.2607
Epoch 56/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2551 -
val_loss: 0.2612
Epoch 57/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2549 -
val_loss: 0.2606
Epoch 58/500
```

```
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2547 -
val_loss: 0.2615
Epoch 59/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2549 -
val_loss: 0.2608
Epoch 60/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2549 -
val_loss: 0.2606
Epoch 61/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2548 -
val_loss: 0.2612
Epoch 62/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2548 -
val_loss: 0.2610
Epoch 63/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2546 -
val_loss: 0.2608
Epoch 64/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2546 -
val_loss: 0.2609
Epoch 65/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2545 -
val_loss: 0.2606
Epoch 66/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2547 -
val_loss: 0.2611
Epoch 67/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2546 -
val_loss: 0.2614
Epoch 68/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2547 -
val_loss: 0.2614
Epoch 69/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2541 -
val_loss: 0.2608
Epoch 70/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2544 -
val_loss: 0.2603
Epoch 71/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2547 -
val_loss: 0.2613
Epoch 72/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2542 -
val_loss: 0.2608
Epoch 73/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2543 -
val_loss: 0.2610
Epoch 74/500
```

```
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2544 -
val_loss: 0.2614
Epoch 75/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2542 -
val_loss: 0.2606
Epoch 76/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2542 -
val_loss: 0.2606
Epoch 77/500
1236/1236 [==============================] - 3s 2ms/step - loss: 0.2540 -
val_loss: 0.2614
Epoch 78/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2538 -
val_loss: 0.2618
Epoch 79/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2538 -
val_loss: 0.2610
Epoch 80/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2541 -
val_loss: 0.2602
Epoch 81/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2540 -
val_loss: 0.2607
Epoch 82/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2540 -
val_loss: 0.2610
Epoch 83/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2537 -
val_loss: 0.2607
Epoch 84/500
1236/1236 [==============================] - 4s 4ms/step - loss: 0.2538 -
val_loss: 0.2616
Epoch 85/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2539 -
val_loss: 0.2617
Epoch 86/500
1236/1236 [==============================] - 4s 4ms/step - loss: 0.2537 -
val_loss: 0.2607
Epoch 87/500
1236/1236 [==============================] - 7s 5ms/step - loss: 0.2534 -
val_loss: 0.2610
Epoch 88/500
1236/1236 [==============================] - 6s 5ms/step - loss: 0.2536 -
val_loss: 0.2608
Epoch 89/500
1236/1236 [==============================] - 6s 4ms/step - loss: 0.2535 -
val_loss: 0.2605
Epoch 90/500
```

```
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2536 -
val_loss: 0.2611
Epoch 91/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2537 -
val_loss: 0.2617
Epoch 92/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2535 -
val_loss: 0.2619
Epoch 93/500
1236/1236 [==============================] - 7s 6ms/step - loss: 0.2539 -
val_loss: 0.2612
Epoch 94/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2534 -
val_loss: 0.2611
Epoch 95/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2536 -
val_loss: 0.2613
Epoch 96/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2533 -
val_loss: 0.2615
Epoch 97/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2534 -
val_loss: 0.2613
Epoch 98/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2534 -
val_loss: 0.2612
Epoch 99/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2534 -
val_loss: 0.2617
Epoch 100/500
1236/1236 [==============================] - 5s 4ms/step - loss: 0.2533 -
val_loss: 0.2611
Epoch 101/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2532 -
val_loss: 0.2615
Epoch 102/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2532 -
val_loss: 0.2623
Epoch 103/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2534 -
val_loss: 0.2614
Epoch 104/500
1236/1236 [==============================] - 4s 3ms/step - loss: 0.2534 -
val_loss: 0.2610
Epoch 105/500
1236/1236 [==============================] - 3s 3ms/step - loss: 0.2532 -
val_loss: 0.2608
Epoch 00105: early stopping
```

[266]: <tensorflow.python.keras.callbacks.History at 0x1be98ef31d0>

[267]:
```
from tensorflow.keras.models import load_model
```

[268]:
```
model.save('mymodel.h5')
```

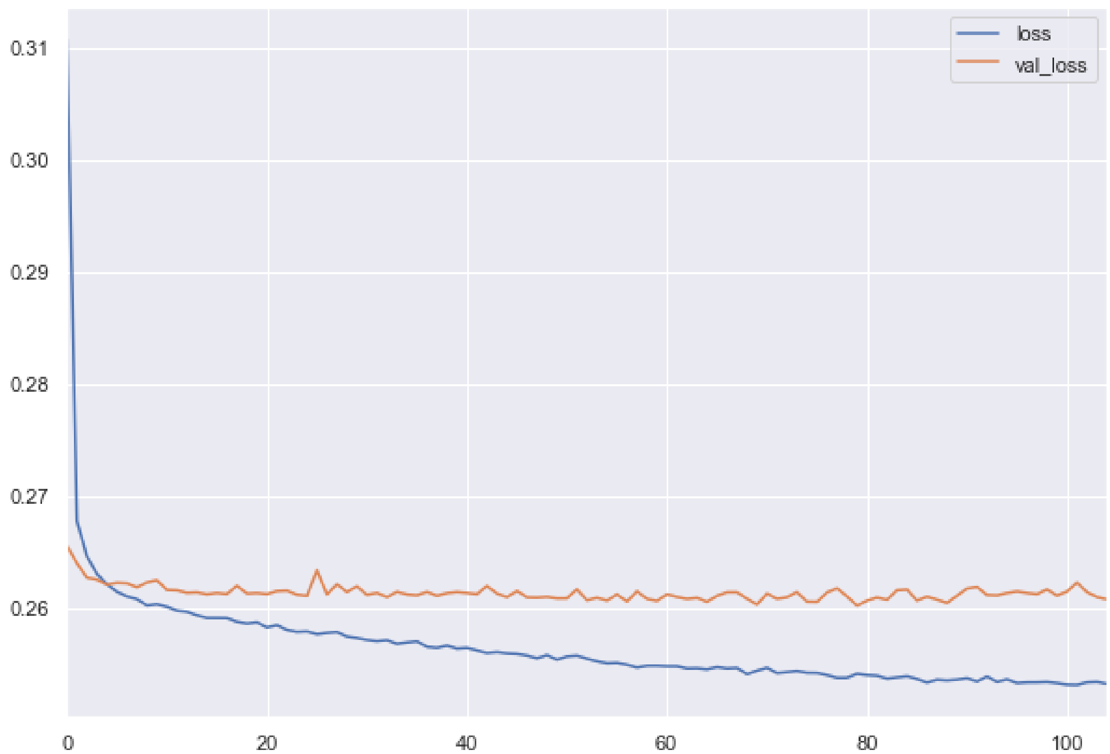
1.6 Evaluating Model Performance.

Let's plot out the validation loss versus the training loss.

[269]:
```
losses = pd.DataFrame(model.history.history)
```

[302]:
```
losses.plot()
```

[302]: <matplotlib.axes._subplots.AxesSubplot at 0x1be96c2fba8>



Now we use our model to reate predictions from the X_test set.

[271]:
```
pred = model.predict_classes(X_test)
```

To check the model performance we will use classification report and confusion matrix.

[272]:
```
from sklearn.metrics import classification_report, confusion_matrix
```

[273]:
```
print(classification_report(y_pred = pred, y_true = y_test))
```

```
              precision    recall  f1-score   support
```

```
           0       0.97      0.45      0.61     15658
           1       0.88      1.00      0.93     63386

    accuracy                           0.89     79044
   macro avg       0.93      0.72      0.77     79044
weighted avg       0.90      0.89      0.87     79044
```

[274]:
```
confusion_matrix(y_true = y_test, y_pred = pred)
```

[274]:
```
array([[ 7022,  8636],
       [  214, 63172]], dtype=int64)
```

1.6.1 Reevaluating on the Full Dataset

[275]:
```
X_full = df.drop('loan_repaid', axis = 1)
y_full = df['loan_repaid']
```

[276]:
```
X_full = scaler.transform(X_full)
```

[277]:
```
pred_full = model.predict_classes(X_full)
```

[278]:
```
print(classification_report(y_pred = pred_full, y_true = y_full))
```

```
              precision    recall  f1-score   support

           0       0.98      0.45      0.62     77523
           1       0.88      1.00      0.94    317696

    accuracy                           0.89    395219
   macro avg       0.93      0.72      0.78    395219
weighted avg       0.90      0.89      0.87    395219
```

[279]:
```
confusion_matrix(y_pred = pred_full, y_true = y_full)
```

[279]:
```
array([[ 35027,  42496],
       [   896, 316800]], dtype=int64)
```

1.7 Final Conclusion

In this project we used Keras library and built Deep Learning Neural Network to predict loan default using LendingClub data. Due to significant imbalance of the target feature (80/20) the least accuracy we expected from the model is 80%.

Our final F1 Score is .89, which is good, but considiring the lower limit of 80%, this score is not fantastic.

Our model did a great job predicting 'Paid Off' loans almost perfectly. However, the model is only 45% accurate when it is trying to predict 'Charged Off' loans.