Australian Credit Approval Analysis

IBM Supervised Machine Learning: Classification

Course Assignment

Vladas Alesius

2021-06-13

Main Objective

The purpose of this analysis is to examine how well different classifier methods are able to distinguish between approved and disapproved credit card applications. We are interested in performance metrics (accuracy, precision, sensitivity etc.) of different methods - we have selected logistic regression, decision trees and random forests. In other words, this analysis will focus on **predictability**.

The analyzed dataset contains information about Australian credit card applications. All attribute names and values have been changed to meaningless symbols to protect confidentiality of the data. The source is this link.

We will start with exploratory data analysis, to see how our variables are distributed and correlated, what transformations might be necessary. Since we will fit regularized logistic regression on our data, it is useful to have normally distributed, scaled features with no outliers.

After initial data pre-processing steps, it will be split into training and testing subsets. Yeo-Johnson transformation and standardization will be fit on the training subset, and applied to the testing subset as well. After that, cross-validation will be used to tune the hyperparameters for each model. The following measures will be assessed for each model:

- Classification report, containing precision, recall, accuracy and F1-score for both application classes
- Area Under (ROC) Curve (AUC) measure
- Confusion matrix
- Average prediction probabilities for each application group in the confusion matrix (true positive, false negative etc.)
- True and false positive rates, and precision and recall values with multiple thresholds, used for model comparison

After that, we will discuss key insights and select the final model. Also, some suggestions about how to improve this study will be presented.

Data Description

We begin with importing necessary modules:

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.model_selection import StratifiedShuffleSplit, GridSearchCV
from sklearn.linear_model import LogisticRegressionCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score, roc_curve, precision_recall
from sklearn.metrics import ConfusionMatrixDisplay, accuracy_score, precision_score, recall_score, f1_score
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.preprocessing import PowerTransformer
from matplotlib import pyplot as plt
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

Let's read the credit dataset and examine the first 5 rows:

In [2]:
```
credit = pd.read_table(r"C:\Users\Lenovo\Desktop\ibm_machine_learning\3_supervised_machine_learning_classificat
credit.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	22.08	11.46	2	4	4	1.585	0	0	0	1	2	100	1213	0
1	0	22.67	7.00	2	8	4	0.165	0	0	0	0	2	160	1	0
2	0	29.58	1.75	1	4	4	1.250	0	0	0	1	2	280	1	0
3	0	21.67	11.50	1	5	3	0.000	1	1	11	1	2	0	1	1
4	1	20.17	8.17	2	6	4	1.960	1	1	14	0	2	60	159	1

In order to process dataset columns, we will assign them names as given in the description above (A1-A15):

In [3]:
```
credit.columns = ['A' + str(i) for i in range (1,16)]
credit.sample(5)
```

Out[3]:

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
159	1	47.83	4.165	2	14	5	0.085	0	0	0	1	2	520	1	0
143	1	64.08	20.000	2	14	8	17.500	1	1	9	1	2	0	1001	1
657	1	39.50	1.625	2	8	4	1.500	0	0	0	0	2	0	317	0
598	1	24.08	0.875	2	7	4	0.085	0	1	4	0	2	254	1951	0
8	1	27.83	1.000	1	2	8	3.000	0	0	0	0	2	176	538	0

The dataset contains 6 numerical and 8 categorical attributes. According to the dataset description, the labels have been changed for the convenience of the statistical algorithms. For example, attribute 4 originally had 3 labels p,g,gg and these have been changed to labels 1,2,3.

- A1: 0,1 CATEGORICAL (formerly: a,b)
- A2: continuous.
- A3: continuous.
- A4: 1,2,3 CATEGORICAL (formerly: p,g,gg)
- A5: 1, 2,3,4,5, 6,7,8,9,10,11,12,13,14 CATEGORICAL (formerly: ff,d,i,k,j,aa,m,c,w, e, q, r,cc, x)
- A6: 1, 2,3, 4,5,6,7,8,9 CATEGORICAL (formerly: ff,dd,j,bb,v,n,o,h,z)
- A7: continuous.
- A8: 1, 0 CATEGORICAL (formerly: t, f)
- A9: 1, 0 CATEGORICAL (formerly: t, f)
- A10: continuous.
- A11: 1, 0 CATEGORICAL (formerly t, f)
- A12: 1, 2, 3 CATEGORICAL (formerly: s, g, p)
- A13: continuous.
- A14: continuous.
- A15: 1,2 class attribute (formerly: +,-) (negative class is presented as 0 instead of 2)

The summary below shows that this dataset contains only numeric (integer or float) variables, and no null values:

In [4]:
```
credit.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 690 entries, 0 to 689
Data columns (total 15 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   A1      690 non-null    int64  
 1   A2      690 non-null    float64
 2   A3      690 non-null    float64
 3   A4      690 non-null    int64  
 4   A5      690 non-null    int64  
 5   A6      690 non-null    int64  
 6   A7      690 non-null    float64
 7   A8      690 non-null    int64  
 8   A9      690 non-null    int64  
 9   A10     690 non-null    int64  
 10  A11     690 non-null    int64  
 11  A12     690 non-null    int64  
 12  A13     690 non-null    int64  
 13  A14     690 non-null    int64  
 14  A15     690 non-null    int64  
dtypes: float64(3), int64(12)
memory usage: 81.0 KB
```

However, as per the dataset description, 37 cases (5%) had one or more missing values. The missing values from particular attributes were:

- A1: 12
- A2: 12
- A4: 6
- A5: 6
- A6: 9
- A7: 9
- A14: 13

These were replaced by the mode of the attribute (categorical) and the mean of the attribute (continuous). The replacement was already done in the source data.

The dataset will be split into two - features (*X*) and the target variable (*y*):

In [5]:
```
X = credit.drop('A15', axis='columns')
y = credit['A15']
```

The class distribution (variable *A15*) is given below, it is (almost) balanced:

```
+: 307 (44.5%)    CLASS 1
-: 383 (55.5%)    CLASS 0
```

As per the dataset description, the necessary features will be converted from integer type to categorical:

In [6]:
```
col_list = ['A1', 'A4', 'A5', 'A6', 'A8', 'A9', 'A11', 'A12']
for col in col_list:
    if X[col].dtype == 'int64':
        X[col] = X[col].astype("category")
```

In [7]:
```
X.dtypes
```

Out[7]:
```
A1     category
A2      float64
A3      float64
A4     category
A5     category
A6     category
A7      float64
A8     category
A9     category
A10       int64
A11    category
A12    category
A13       int64
A14       int64
dtype: object
```

Data Exploration

As per this source, performing the following steps might improve the accuracy of logistic regression model:

- Removing potential outliers
- Making sure that the predictor variables are normally distributed. If not, you can use data transformations.
- Remove highly correlated predictors to minimize overfitting. The presence of highly correlated predictors might lead to an unstable model solution.

The following summary of the numeric variables shows that some of them might be skewed and have outliers - maximum values are large, compared to other parameters, and the standard deviations are often bigger than the means. For example, A14 has a five times bigger standard deviation than its mean:

In [8]:
```
X.describe()
```

Out[8]:

	A2	A3	A7	A10	A13	A14
count	690.000000	690.000000	690.000000	690.00000	690.000000	690.000000
mean	31.568203	4.758725	2.223406	2.40000	184.014493	1018.385507
std	11.853273	4.978163	3.346513	4.86294	172.159274	5210.102598
min	13.750000	0.000000	0.000000	0.00000	0.000000	1.000000
25%	22.670000	1.000000	0.165000	0.00000	80.000000	1.000000
50%	28.625000	2.750000	1.000000	0.00000	160.000000	6.000000
75%	37.707500	7.207500	2.625000	3.00000	272.000000	396.500000
max	80.250000	28.000000	28.500000	67.00000	2000.000000	100001.000000

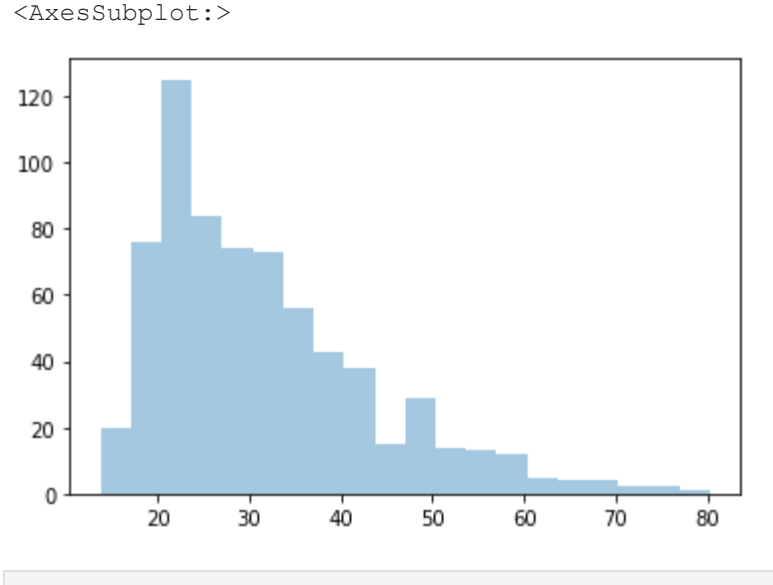
The histograms below confirm that the numeric features are heavily skewed and have some outliers. The skew will be later dealt with Yeo-Johnson transformation and data standardization. However, the outliers will be kept in the dataset, since we are unaware of their nature.

Scatterplots demonstrate the differences in feature distribution for each application class. The biggest difference is for variable A14 - class 1 values vary much more, the outliers are very large. This might negatively impact the predictive power for class 1 cases.

In [32]:
```
sns.distplot(x=credit['A2'], kde=False)
```

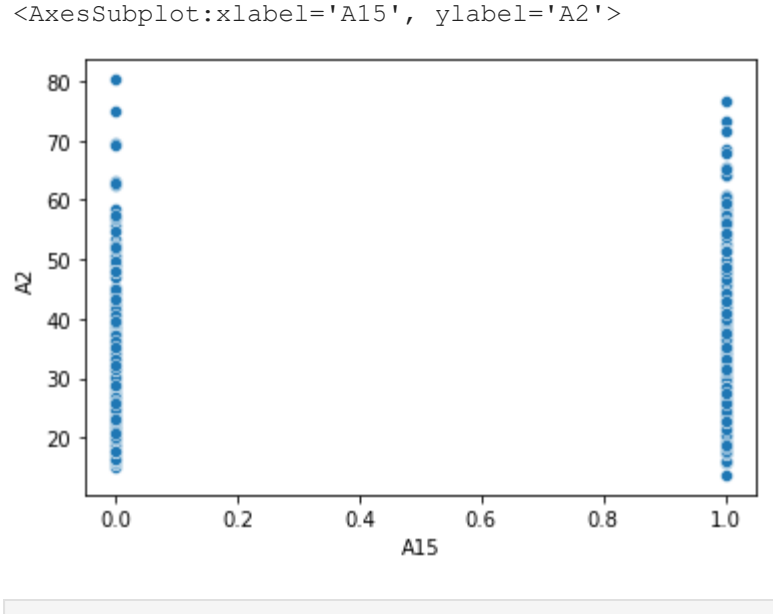
```
C:\Users\Lenovo\anaconda3\lib\site-packages\seaborn\distributions.py:2551: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[32]: <AxesSubplot:>



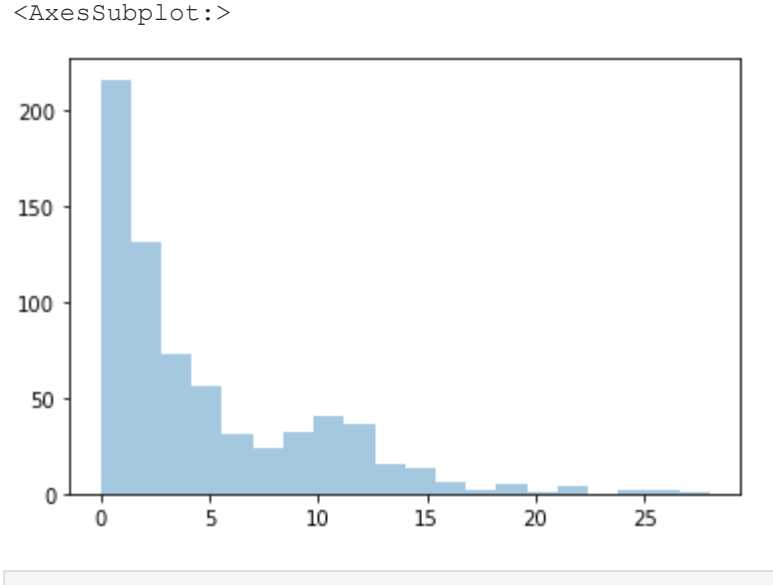
In [42]:
```
sns.scatterplot(data=credit, x='A15', y='A2')
```

Out[42]: <AxesSubplot:xlabel='A15', ylabel='A2'>



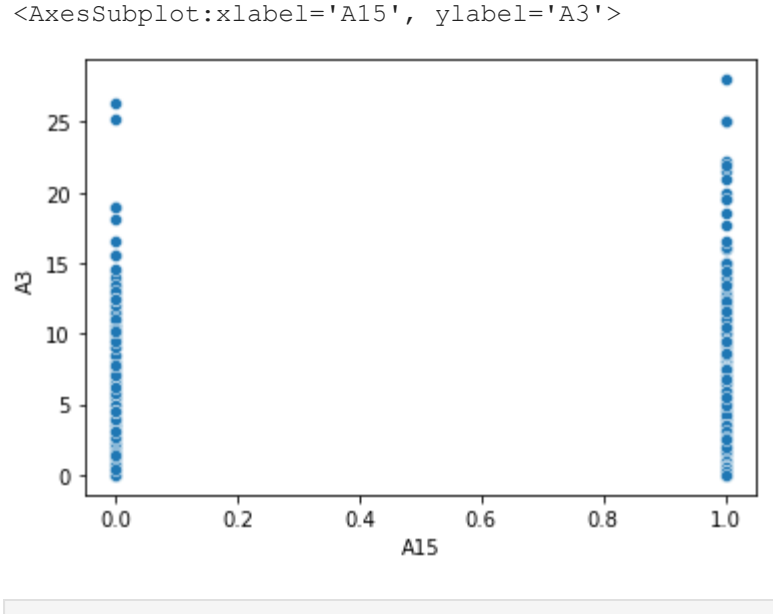
In [33]:
```
sns.distplot(x=credit['A3'], kde=False)
```

Out[33]: <AxesSubplot:>



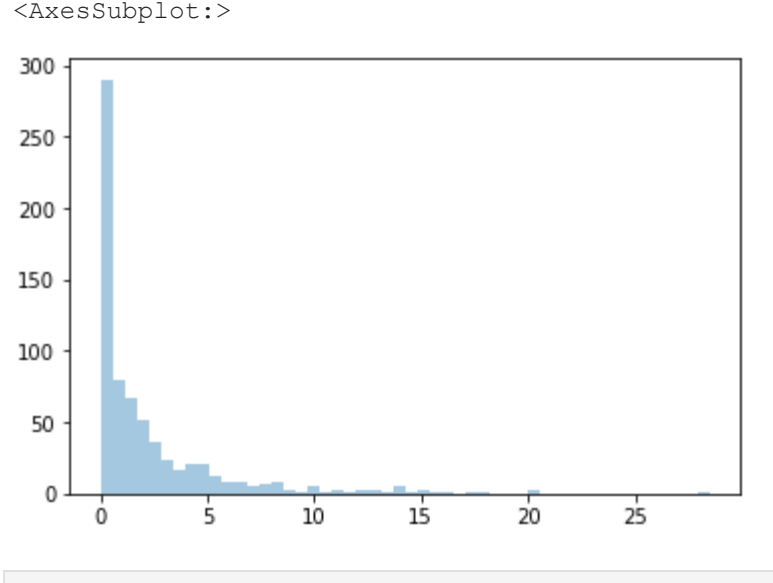
In [41]:
```
sns.scatterplot(data=credit, x='A15', y='A3')
```

Out[41]: <AxesSubplot:xlabel='A15', ylabel='A3'>



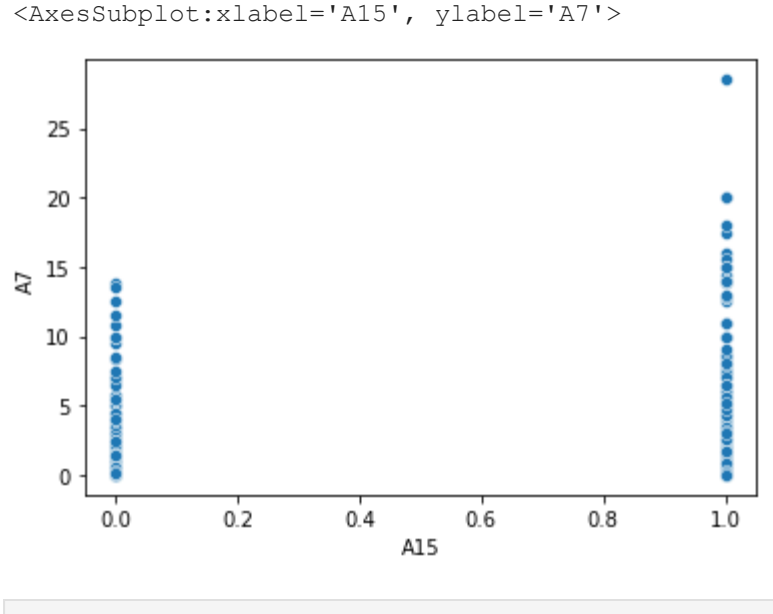
In [34]:
```
sns.distplot(x=credit['A7'], kde=False)
```

Out[34]: <AxesSubplot:>



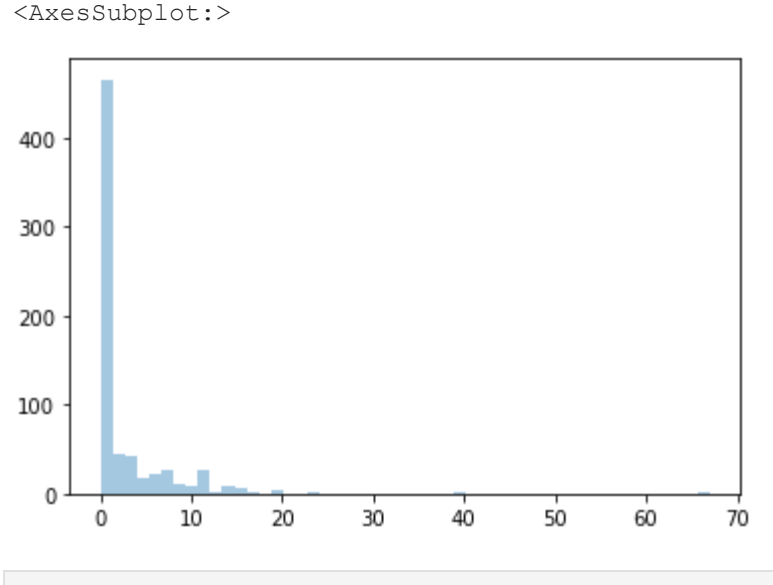
In [40]:
```
sns.scatterplot(data=credit, x='A15', y='A7')
```

Out[40]: <AxesSubplot:xlabel='A15', ylabel='A7'>



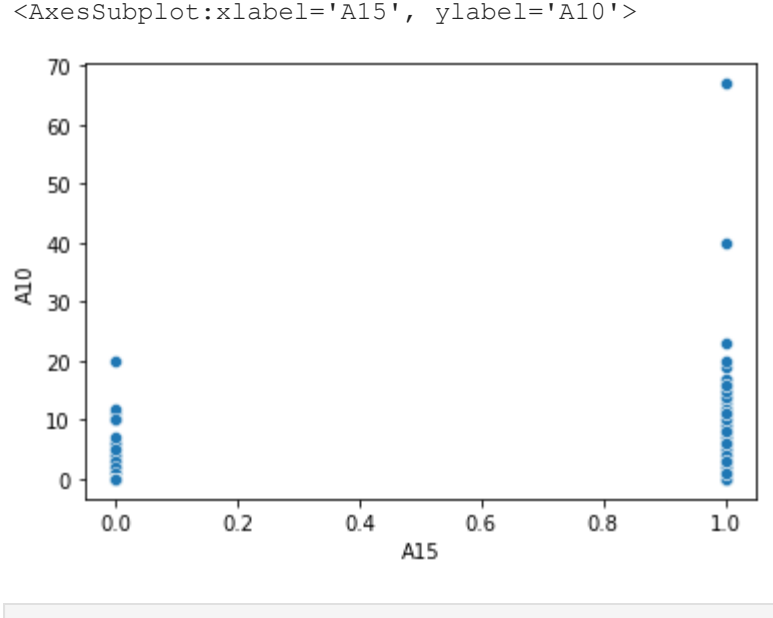
In [35]:
```
sns.distplot(x=credit['A10'], kde=False)
```

Out[35]: <AxesSubplot:>



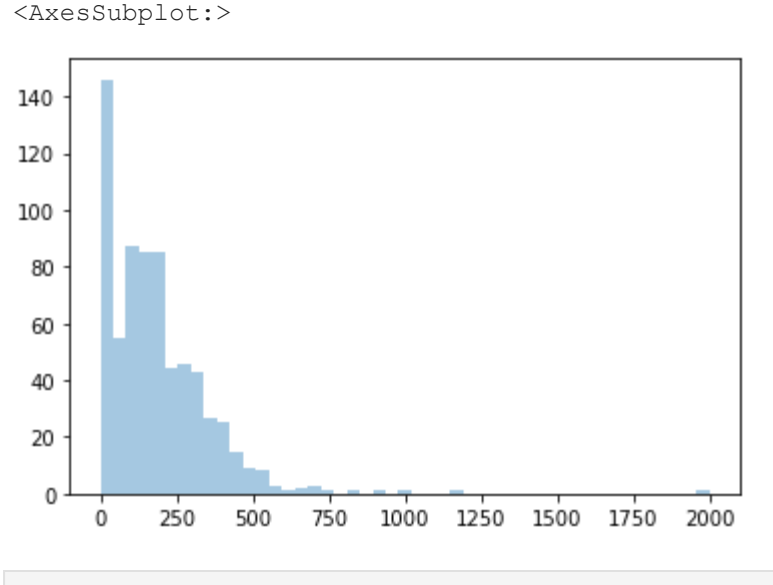
In [39]:
```
sns.scatterplot(data=credit, x='A15', y='A10')
```

Out[39]: <AxesSubplot:xlabel='A15', ylabel='A10'>



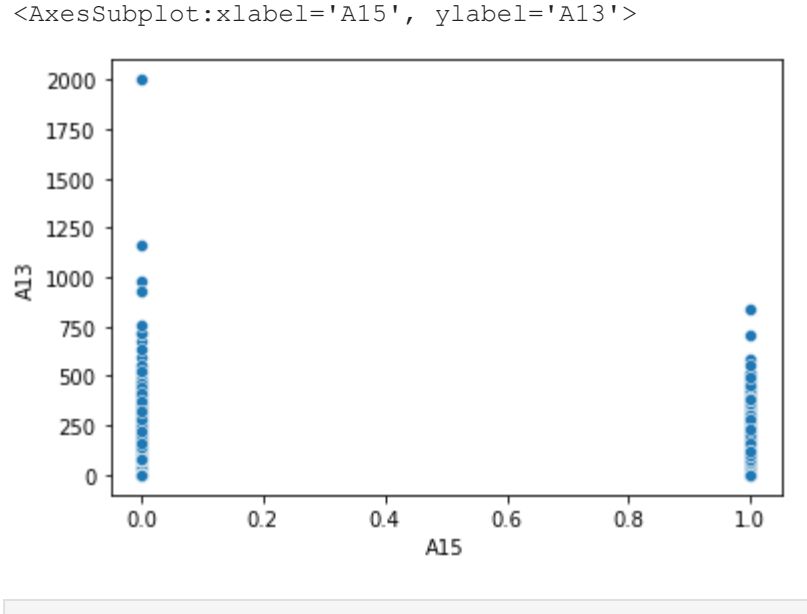
In [36]:
```
sns.distplot(x=credit['A13'], kde=False)
```

Out[36]: <AxesSubplot:>



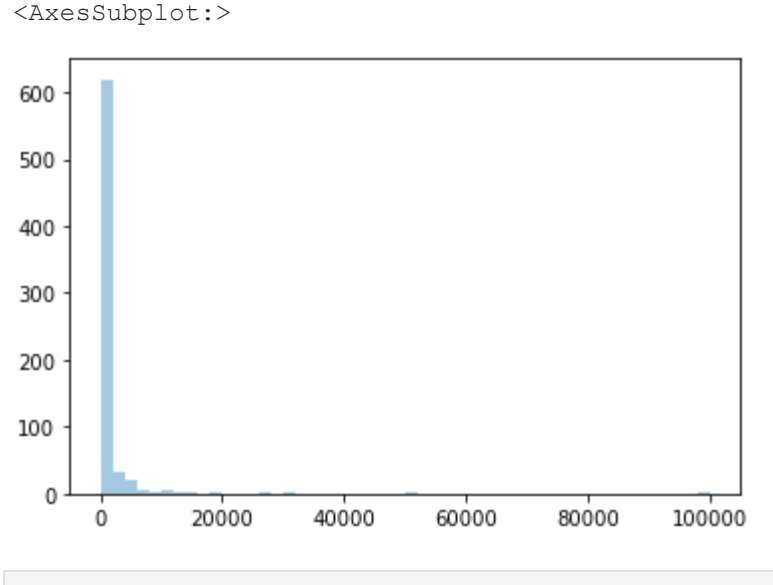
In [38]:
```
sns.scatterplot(data=credit, x='A15', y='A13')
```

Out[38]: <AxesSubplot:xlabel='A15', ylabel='A13'>



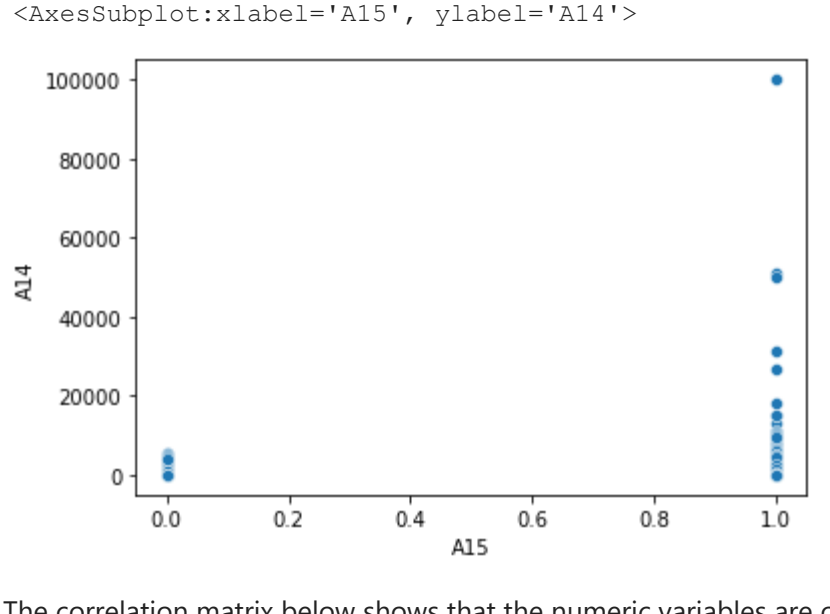
In [37]:
```
sns.distplot(x=credit['A14'], kde=False)
```

Out[37]: <AxesSubplot:>



In [27]:
```
sns.scatterplot(data=credit, x='A15', y='A14')
```

Out[27]: <AxesSubplot:xlabel='A15', ylabel='A14'>



The correlation matrix below shows that the numeric variables are only weakly correlated, so it is unnecessary to remove any of them:

In [15]:
```
X.corr()
```

Out[15]:

	A2	A3	A7	A10	A13	A14
A2	1.000000	0.201315	0.392788	0.185574	-0.077159	0.018539
A3	0.201315	1.000000	0.298902	0.271207	-0.222346	0.123121
A7	0.392788	0.298902	1.000000	0.322330	-0.076389	0.051345
A10	0.185574	0.271207	0.322330	1.000000	-0.119808	0.063692
A13	-0.077159	-0.222346	-0.076389	-0.119808	1.000000	0.065609
A14	0.018539	0.123121	0.051345	0.063692	0.065609	1.000000

The next step is to convert categorical variables into dummies - variables with 0/1 value for each category (except the first value in order to avoid colinearity):

In [16]:
```
X = pd.get_dummies(X, drop_first=True)
```

The numeric variables remain as they were before, and the categorical variables are now converted. The total number of features increased from 14 to 34:

In [17]:
```
X
```

Out[17]:

	A2	A3	A7	A10	A13	A14	A1_1	A4_2	A4_3	A5_2	...	A6_4	A6_5	A6_7	A6_8	A6_9	A8_1	A9_1	A11_1	A12_2	A12_3
0	22.08	11.460	1.585	0	100	1213	1	1	0	0	...	1	0	0	0	0	0	0	1	1	0
1	22.67	7.000	0.165	0	160	1	0	1	0	0	...	1	0	0	0	0	0	0	0	1	0
2	29.58	1.750	1.250	0	280	1	0	0	0	0	...	1	0	0	0	0	0	0	1	1	0
3	21.67	11.500	0.000	11	0	1	0	0	0	0	...	0	0	0	0	0	1	1	1	1	0
4	20.17	8.170	1.960	14	60	159	1	1	0	0	...	1	0	0	0	0	1	1	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
685	31.57	10.500	6.500	0	0	1	1	1	0	0	...	1	0	0	0	0	1	0	0	1	0
686	20.67	0.415	0.125	0	0	45	1	1	0	0	...	1	0	0	0	0	0	0	0	1	0
687	18.83	9.540	0.085	0	100	1	0	1	0	0	...	1	0	0	0	0	1	0	0	1	0
688	27.42	14.500	3.085	1	120	12	0	1	0	0	...	0	0	0	1	0	1	1	0	1	0
689	41.00	0.040	0.040	1	560	1	1	1	0	0	...	1	0	0	0	0	0	1	0	0	0

690 rows × 34 columns

Classification Models

Before trying the models, some additional preparation steps are needed.

The first step is to split the dataset into training and testing subsets. Scikit-learn's `StratifiedShuffleSplit` will be used to maintain the same ratio of predictor classes. 30% of the data will be used for testing:

In [18]:
```
# Get the split indexes
strat_shuf_split = StratifiedShuffleSplit(n_splits=1,
                                          test_size=0.3,
                                          random_state=101)

train_idx, test_idx = next(strat_shuf_split.split(X, y))

# Create the dataframes
X_train = X.loc[train_idx, ]
y_train = y.loc[train_idx, ]

X_test  = X.loc[test_idx, ]
y_test  = y.loc[test_idx, ]
```

Training and testing subsets have 483 and 207 rows respectively:

In [19]:
```
X_train.shape
```

Out[19]: (483, 34)

In [20]:
```
X_test.shape
```

Out[20]: (207, 34)

The comparison below shows that stratified split keeps class proportions:

In [21]:
```
y_train.value_counts(normalize=True)
```

Out[21]:
```
0    0.554865
1    0.445135
Name: A15, dtype: float64
```

In [22]:
```
y_test.value_counts(normalize=True)
```

Out[22]:
```
0    0.555556
1    0.444444
Name: A15, dtype: float64
```

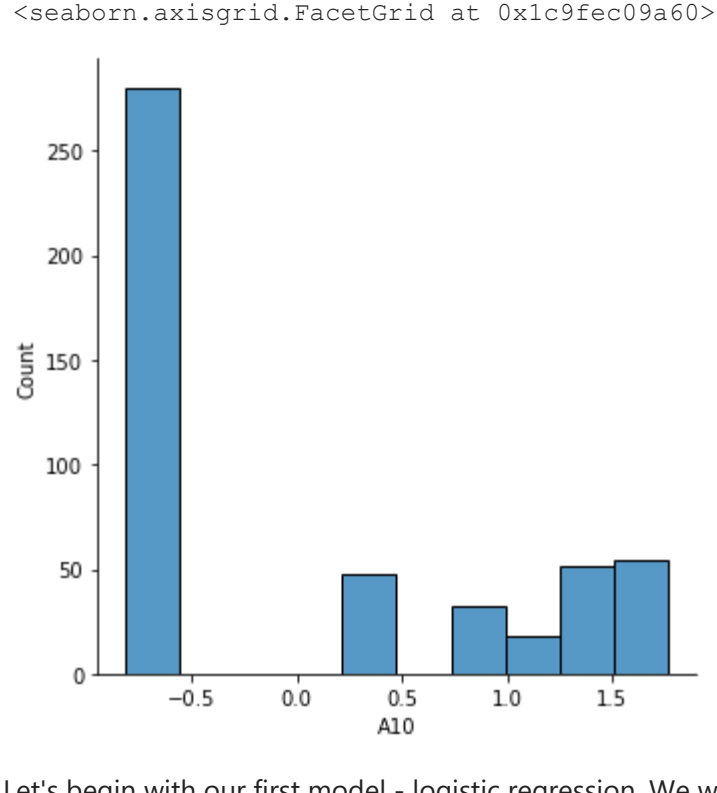
One of the models that we will examine is logistic regression, its efficiency is improved by scaling. As per the exploratory data analysis above, our predictors are heavily skewed, so we will apply Yeo-Johnson transformation together with standardization using Scikit's `PowerTransformer` class to make the predictors as similar to standard normal distribution as possible. This transformation will be fit on the training subset to avoid data leakage, and will be used to transform the testing dataset as well. Also, to be able to compare the analyzed models (logistic regression, decision tree, random forest), the transformed datasets will be used for all of them:

In [23]:
```
s = PowerTransformer()
X_train_s = s.fit_transform(X_train)
X_train_s = pd.DataFrame(X_train_s, columns=X.columns)
X_test_s = s.transform(X_test)
X_test_s = pd.DataFrame(X_test_s, columns=X.columns)
```

However, the following histogram shows that not all variables became bell shaped after the transformation. So the transformation might be not as useful for model efficiency as one could expect:

In [24]:
```
sns.displot(X_train_s['A10'])
```

Out[24]: <seaborn.axisgrid.FacetGrid at 0x1c9fec09a60>



Let's begin with our first model - logistic regression. We will perform cross validation estimation for the best C value. According to Scikit-learn User Guide, `liblinear` is a good solver choice for small datasets. Otherwise, we will stick to defaults - 10 regularization strength values, L2 regularization and 5-fold cross validation:

In [25]:
```
lr_l2 = LogisticRegressionCV(solver='liblinear', random_state=101).fit(X_train_s, y_train)
```

Class predictions are made for both subsets, and prediction probabilities are computed for the testing sample:

In [26]:
```
y_train_pred_lr = lr_l2.predict(X_train_s)
```

In [27]:
```
y_test_pred_lr = lr_l2.predict(X_test_s)
```

In [28]:
```
prob_lr = lr_l2.predict_proba(X_test_s)
```

The output below contains classification metrics for training and testing subsets respectively:

In [29]:
```
print(classification_report(y_train, y_train_pred_lr, digits=4))
```

```
              precision    recall  f1-score   support

           0     0.9157    0.8918    0.9036       268
           1     0.8694    0.8977    0.8833       215

    accuracy                         0.8944       483
   macro avg     0.8925    0.8947    0.8934       483
weighted avg     0.8951    0.8944    0.8946       483
```

In [30]:
```
print(classification_report(y_test, y_test_pred_lr, digits=4))
```

```
              precision    recall  f1-score   support

           0     0.8772    0.8696    0.8734       115
           1     0.8387    0.8478    0.8432        92

    accuracy                         0.8599       207
   macro avg     0.8580    0.8587    0.8583       207
weighted avg     0.8601    0.8599    0.8600       207
```

In [82]:
```
acc_lr = accuracy_score(y_test, y_test_pred_lr)
prec_lr = precision_score(y_test, y_test_pred_lr)
rec_lr = recall_score(y_test, y_test_pred_lr)
f1_lr = f1_score(y_test, y_test_pred_lr)
```

As expected, all metric values are lower for the testing subset, although still pretty good (all exceeding 0.83). For the testing subset, all metrics are better for class 0 instances. Additionally, AUC value is given below:

In [31]:
```
auc_lr = roc_auc_score(y_test, prob_lr[:, 1])
auc_lr
```

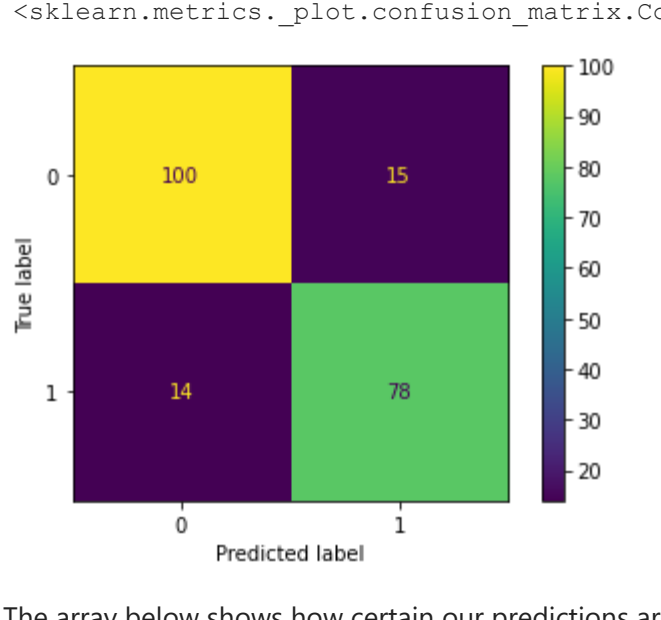
Out[31]: 0.9170132325141778

In [32]:
```
fpr_lr, tpr_lr, _ = roc_curve(y_test, prob_lr[:, 1])
precision_lr, recall_lr, _ = precision_recall_curve(y_test, prob_lr[:, 1])
```

The confusion matrix below confirms that prediction for class 0 is more effective:

In [33]:
```
cm = confusion_matrix(y_test, y_test_pred_lr)
ConfusionMatrixDisplay(cm).plot()
```

Out[33]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c98001b640>

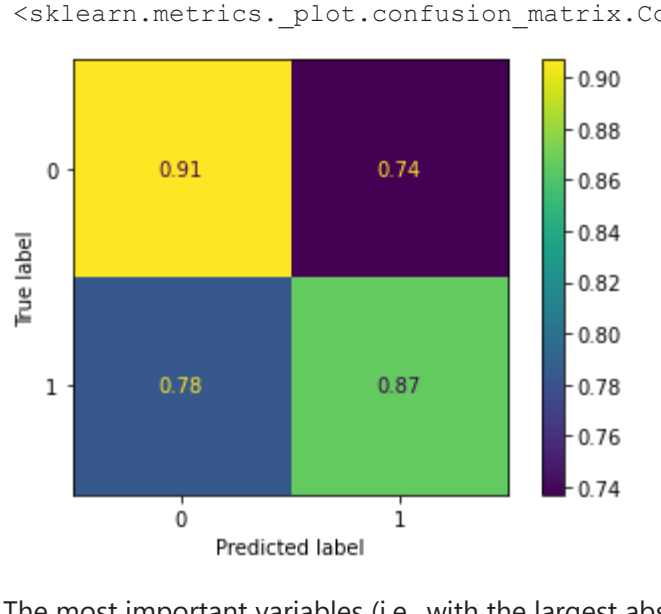


The array below shows how certain our predictions are for each confusion matrix entry. Average probabilities of predictions are much higher for true positives and true negatives:

In [34]:
```
prob_true_0_pred_1 = np.round(np.mean(np.max(prob_lr[(y_test==0) & (y_test_pred_lr==1), :], axis=1)), 3)
prob_true_1_pred_1 = np.round(np.mean(np.max(prob_lr[(y_test==1) & (y_test_pred_lr==1), :], axis=1)), 3)
prob_true_0_pred_0 = np.round(np.mean(np.max(prob_lr[(y_test==0) & (y_test_pred_lr==0), :], axis=1)), 3)
prob_true_1_pred_0 = np.round(np.mean(np.max(prob_lr[(y_test==1) & (y_test_pred_lr==0), :], axis=1)), 3)
```

In [35]:
```
prob_cm = np.array([[prob_true_0_pred_0, prob_true_0_pred_1], [prob_true_1_pred_0, prob_true_1_pred_1]])
ConfusionMatrixDisplay(prob_cm).plot()
```

Out[35]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c9800acf40>



The most important variables (i.e., with the largest absolute coefficient values) are given below. A8_1 variable stands out, several other important variables are the numeric ones, others are dummy variables:

In [36]:
```
lr_l2_coef = pd.DataFrame(zip(X.columns, lr_l2.coef_[0])).sort_values(by=1)
lr_l2_coef[abs(lr_l2_coef.iloc[:, 1]) > 0.3]
```

Out[36]:

	0	1
4	A13	-0.474748
7	A4_2	0.320030
16	A5_9	0.328330
27	A6_8	0.335755
21	A5_14	0.402530
5	A14	0.424243
2	A7	0.453842
3	A10	0.494117
29	A8_1	1.673964

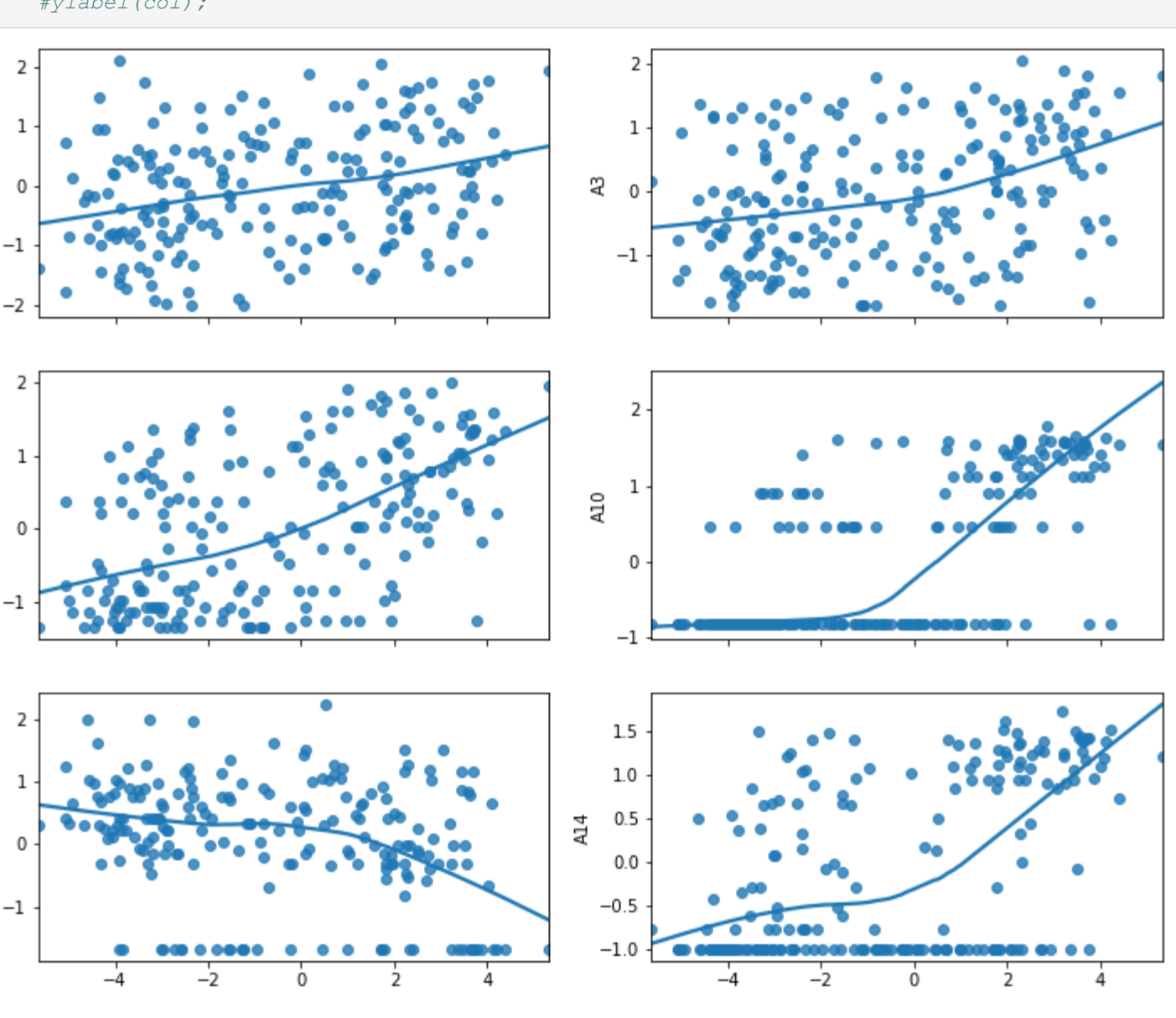
The final step is to examine whether the following logistic regression assumptions hold true for the data:

- There is a linear relationship between the logit of the outcome and each predictor variables
- There are no influential values (extreme values or outliers) in the continuous predictors
- There are no high intercorrelations (i.e. multicollinearity) among the predictors.

Below the logit of the outcome is plotted against each numeric feature:

In [37]:
```
log_odds = lr_l2.decision_function(X_test_s)
```

In [38]:
```
fig, ax = plt.subplots(3, 2, figsize=(12,10), sharex=True)
for i in range(6):
    ax = ax.flatten()
    plt.sca(ax[i])
    col = X_test_s.columns[i]
    sns.regplot(x=log_odds, y=X_test_s.iloc[:, i], lowess=True)
    #ylabel(col);
```



As we can see, the relationship is non-linear for A10 and A14. Probably using polynomial features could improve model's quality.

The plots in exploratory data analysis showed that there are some outlier values for the numeric features, such as 100001 for A14. Excluding these entries might improve the predictability, but we need to know more about their nature (and actual feature meanings) to do this.

The presence of multicollinearity between the features will be measured using Variance Inflation Factor (VIF). As per this source, "*It is a statistical concept that indicates the increase in the variance of a regression coefficient as a result of collinearity.*" When VIF value exceeds 10, there is significant multicollinearity that needs to be corrected. Below we will check which variables have high multicollinearity:

In [39]:
```
vif_values = [variance_inflation_factor(np.asmatrix(X_train_s), i) for i in range(X_train_s.shape[1])]
vif_df = pd.DataFrame([X_train_s.columns, vif_values]).T
vif_df.columns = ["Feature", "VIF"]
vif_df.sort_values(by=['VIF'], ascending=False).loc[vif_df['VIF'] >= 10, ]
```

Out[39]:

	Feature	VIF
24	A6_4	49.896
15	A5_8	38.1908
27	A6_8	32.4117
18	A5_11	23.7255
16	A5_9	19.6828
25	A6_5	16.8184
10	A5_3	16.7483
13	A5_6	16.3794
3	A10	14.3769
30	A9_1	13.6872
11	A5_4	12.9034
14	A5_7	12.1904
21	A5_14	11.7579
20	A5_13	11.7293
9	A5_2	11.3107
17	A5_10	10.816

Most features with high multicollinearity are dummy variables that represent more than two categories. As per the source quoted above, multicollinearity does not necessarily exist in this case. However, features A10 (numeric) and A9_1 (which has only 2 categories) have high multicollinearities. So these variables could be removed, since the information they provide is redundant.

Decision Tree

The next model we are going to try is decision tree. Firstly, we will fit a decision tree classifier with no set limits on maximum depth, features, or leaves.

In [40]:
```
dt = DecisionTreeClassifier(random_state=101)
dt = dt.fit(X_train_s, y_train)
```

It has 111 nodes and its depth is 13:

In [41]:
```
dt.tree_.node_count, dt.tree_.max_depth
```

Out[41]: (111, 13)

Predictions for both subsets will be made using the first decision tree classifier:

In [42]:
```
y_train_pred = dt.predict(X_train_s)
y_test_pred = dt.predict(X_test_s)
```

All metrics equal 1 for the training subset, and are significantly lower for the testing subset, which indicates overfitting.

In [43]:
```
print(classification_report(y_train, y_train_pred, digits=4))
```
```
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000       268
           1     1.0000    1.0000    1.0000       215

    accuracy                         1.0000       483
   macro avg     1.0000    1.0000    1.0000       483
weighted avg     1.0000    1.0000    1.0000       483
```

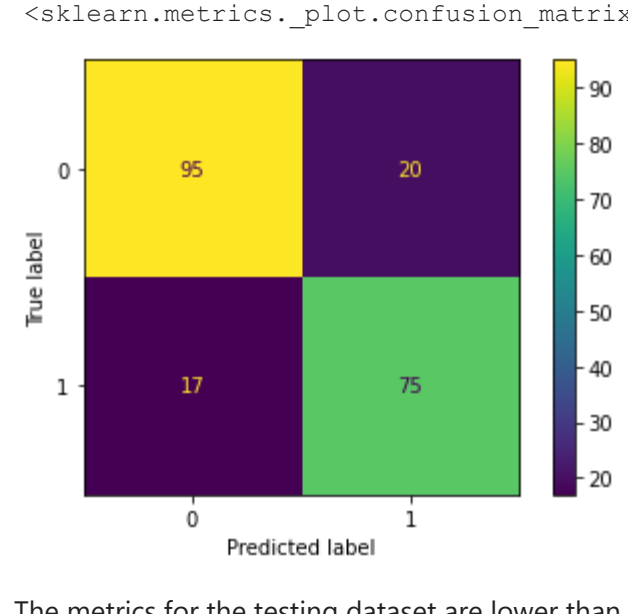
In [44]:
```
print(classification_report(y_test, y_test_pred, digits=4))
```
```
              precision    recall  f1-score   support

           0     0.8482    0.8261    0.8370       115
           1     0.7895    0.8152    0.8021        92

    accuracy                         0.8213       207
   macro avg     0.8188    0.8207    0.8196       207
weighted avg     0.8221    0.8213    0.8215       207
```

In [45]:
```
ConfusionMatrixDisplay(confusion_matrix(y_test, y_test_pred)).plot()
```

Out[45]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c9815a74c0>



The metrics for the testing dataset are lower than for the logistic regression model, so we will apply grid search to find a tree that will perform better. Maximum tree depth and maximum number of features considered when splitting the node will be the hyperparameters that we will tune the model on. The default 5-fold cross validation will be used for the selection:

In [46]:
```
param_grid = {'max_depth':range(1, dt.tree_.max_depth+1, 2),
              'max_features': range(1, len(dt.feature_importances_)+1)}

best_tree = GridSearchCV(DecisionTreeClassifier(random_state=101),
                 param_grid=param_grid,
                 scoring='accuracy',
                 n_jobs=-1)

best_tree = best_tree.fit(X_train_s, y_train)
```

The number of nodes and tree depth for the best tree are given below. As we can see, the tree is much simpler now:

In [47]:
```
best_tree.best_estimator_.tree_.node_count, best_tree.best_estimator_.tree_.max_depth
```

Out[47]: (3, 1)

Here are the most important features of the pruned tree. The same as for the logistic regression model, A8_1 is the most important (and the only) feature:

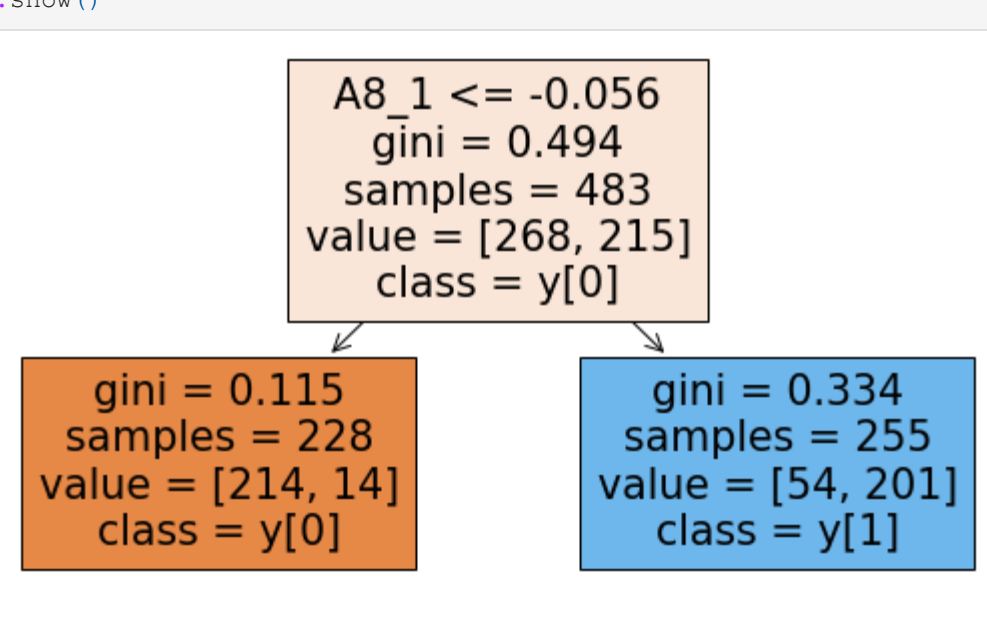
In [48]:
```
tree_coef = pd.DataFrame(zip(X.columns, best_tree.best_estimator_.feature_importances_)).sort_values(by=1)
tree_coef[abs(tree_coef.iloc[:, 1]) > 0]
```

Out[48]:

	0	1
29	A8_1	1.0

The pruned tree is plotted below. The left leaf node corresponding to class 0 is much more pure:

In [49]:
```
f = plt.figure()
f.set_figwidth(10)
f.set_figheight(5)
tree.plot_tree(best_tree.best_estimator_, feature_names=X.columns, filled=True, class_names=True)
plt.show()
```



In [50]:
```
y_train_pred_tree = best_tree.predict(X_train_s)
y_test_pred_tree = best_tree.predict(X_test_s)
prob_tree = best_tree.predict_proba(X_test_s)
```

The classification reports for training and testing datasets show that most metrics are slightly worse for the testing dataset, but they are higher (at least on average) than in case of the 'full' decision tree discussed above:

In [51]:
```
print(classification_report(y_train, y_train_pred_tree, digits=4))
```
```
              precision    recall  f1-score   support

           0     0.9386    0.7985    0.8629       268
           1     0.7882    0.9349    0.8553       215

    accuracy                         0.8592       483
   macro avg     0.8634    0.8667    0.8591       483
weighted avg     0.8717    0.8592    0.8595       483
```

In [52]:
```
print(classification_report(y_test, y_test_pred_tree, digits=4))
```
```
              precision    recall  f1-score   support

           0     0.9109    0.8000    0.8519       115
           1     0.7830    0.9022    0.8384        92

    accuracy                         0.8454       207
   macro avg     0.8470    0.8511    0.8451       207
weighted avg     0.8541    0.8454    0.8459       207
```

In [83]:
```
acc_tree = accuracy_score(y_test, y_test_pred_tree)
prec_tree = precision_score(y_test, y_test_pred_tree)
rec_tree = recall_score(y_test, y_test_pred_tree)
f1_tree = f1_score(y_test, y_test_pred_tree)
```

The decision tree model has a higher precision for class 0 (and lower for class 1), compared to logistic regression. Aggregated metrics are slightly better for logistic regression. Area Under Curve is significantly lower for the decision tree model than for logistic regression:

In [53]:
```
auc_tree = roc_auc_score(y_test, prob_tree[:, 1])
auc_tree
```

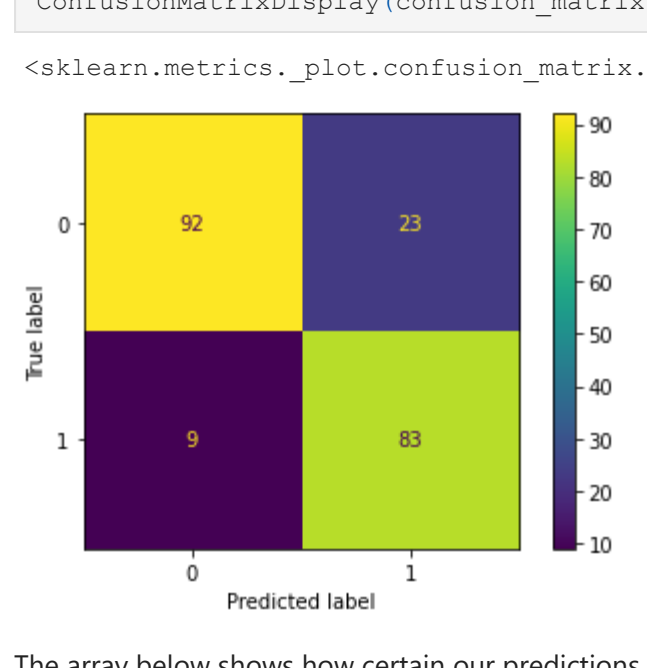
Out[53]: 0.8510869565217392

In [76]:
```
fpr_tree, tpr_tree, _ = roc_curve(y_test, prob_tree[:, 1])
precision_tree, recall_tree, _ = precision_recall_curve(y_test, prob_tree[:, 1])
```

Decision tree gives more incorrect predictions of class 1 and fewer incorrect predictions of class 0 than logistic regression:

In [55]:
```
ConfusionMatrixDisplay(confusion_matrix(y_test, y_test_pred_tree)).plot()
```

Out[55]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c9813e2d60>

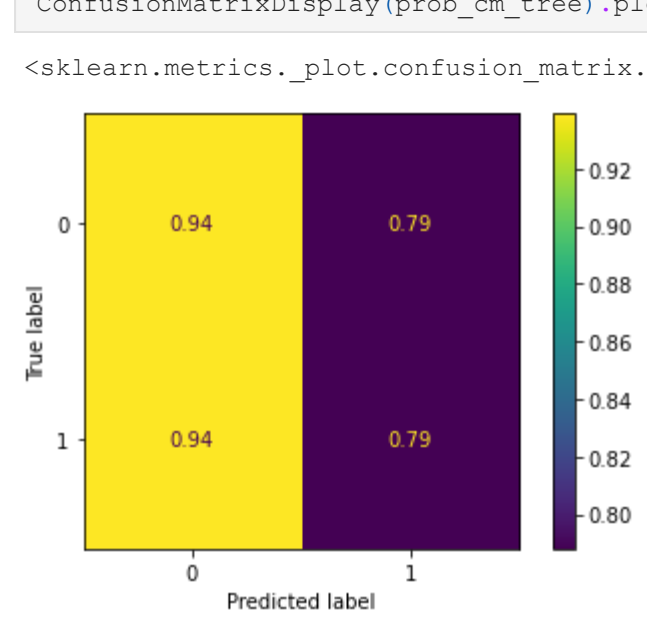


The array below shows how certain our predictions are for each confusion matrix entry. We assign the same probability to each entry in the leaf node, so we have no difference between true and false predictions:

In [56]:
```
prob_true_0_pred_1_tree = np.round(np.mean(np.max(prob_tree[(y_test==0) & (y_test_pred_tree==1), :], axis=1)),
prob_true_1_pred_1_tree = np.round(np.mean(np.max(prob_tree[(y_test==1) & (y_test_pred_tree==1), :], axis=1)),
prob_true_0_pred_0_tree = np.round(np.mean(np.max(prob_tree[(y_test==0) & (y_test_pred_tree==0), :], axis=1)),
prob_true_1_pred_0_tree = np.round(np.mean(np.max(prob_tree[(y_test==1) & (y_test_pred_tree==0), :], axis=1)),
```

In [57]:
```
prob_cm_tree = np.array([[prob_true_0_pred_0_tree, prob_true_0_pred_1_tree],
                         [prob_true_1_pred_0_tree, prob_true_1_pred_1_tree]])
ConfusionMatrixDisplay(prob_cm_tree).plot()
```

Out[57]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c981211310>



Random Forest

The last examined model is random forest. It fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. As for previous models, we select a random state to keep the analysis reproducible. Furthermore, we reuse the solution of the previous call to fit and add more estimators to the ensemble. Also, we use all processors to run the job.

The best (in terms of accuracy) number of trees for the created random forest classifier will be searched using GridSearchCV:

In [58]:
```
RF = RandomForestClassifier(random_state=101,
                            warm_start=True,
                            n_jobs=-1)

param_grid = {'n_estimators':[10, 20, 30, 40, 50, 100, 200, 300, 400, 500]}

best_forest = GridSearchCV(RF,
                 param_grid=param_grid,
                 scoring='accuracy',
                 n_jobs=-1)

best_forest = best_forest.fit(X_train_s, y_train)
```

The most accurate forest consists of 300 trees and has an accuracy of about 0.87:

In [59]:
```
best_forest.best_score_
```

Out[59]: 0.8715850515463919

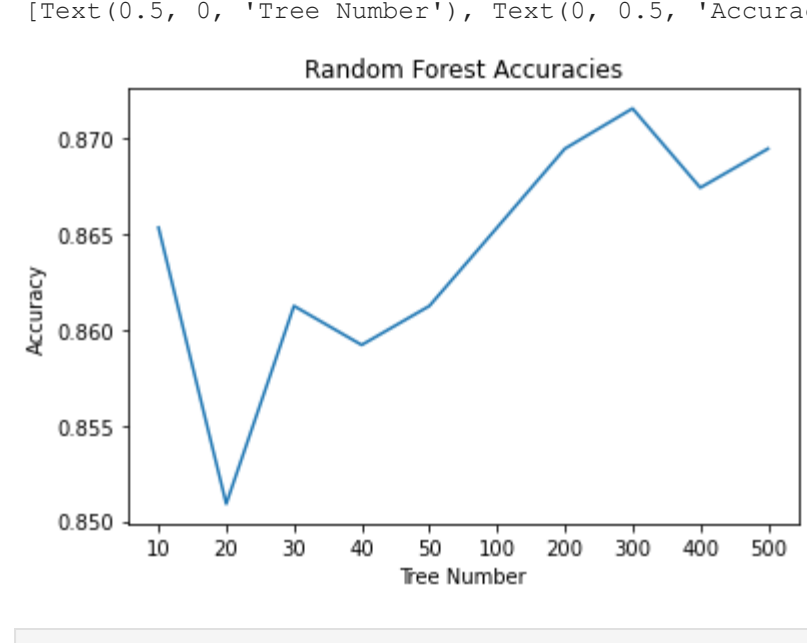
In [60]:
```
best_forest.best_params_
```

Out[60]: {'n_estimators': 300}

The accuracies of other forests are not much different, above 0.85:

In [61]:
```
g = sns.lineplot(x=range(len(param_grid['n_estimators'])), y=best_forest.cv_results_['mean_test_score'])
g.set_xticks(range(len(param_grid['n_estimators'])))
g.set_xticklabels(param_grid['n_estimators'])
g.set_title("Random Forest Accuracies")
g.set(xlabel='Tree Number', ylabel='Accuracy')
```

Out[61]: [Text(0.5, 0, 'Tree Number'), Text(0, 0.5, 'Accuracy')]



In [62]:
```
y_train_pred_forest = best_forest.predict(X_train_s)
y_test_pred_forest = best_forest.predict(X_test_s)
prob_forest = best_forest.predict_proba(X_test_s)
```

The most important features are listed below. A8_1 is still the most important, followed by the numeric features:

In [63]:
```
forest_coef = pd.DataFrame(zip(X.columns, best_forest.best_estimator_.feature_importances_)).sort_values(by=1)
forest_coef[abs(forest_coef.iloc[:, 1]) > 0.05]
```

Out[63]:

	0	1
30	A9_1	0.050840
4	A13	0.067455
0	A2	0.068047
1	A3	0.072585
3	A10	0.086647
5	A14	0.087241
2	A7	0.114104
29	A8_1	0.269123

Although all metric values are 1 for the training subset (which might indicate overfitting), the testing metrics are high as well:

In [64]:
```
print(classification_report(y_train, y_train_pred_forest, digits=4))
```
```
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000       268
           1     1.0000    1.0000    1.0000       215

    accuracy                         1.0000       483
   macro avg     1.0000    1.0000    1.0000       483
weighted avg     1.0000    1.0000    1.0000       483
```

In [65]:
```
print(classification_report(y_test, y_test_pred_forest, digits=4))
```
```
              precision    recall  f1-score   support

           0     0.8696    0.8696    0.8696       115
           1     0.8370    0.8370    0.8370        92

    accuracy                         0.8551       207
   macro avg     0.8533    0.8533    0.8533       207
weighted avg     0.8551    0.8551    0.8551       207
```

In [91]:
```
acc_forest = accuracy_score(y_test, y_test_pred_forest)
prec_forest = precision_score(y_test, y_test_pred_forest)
rec_forest = recall_score(y_test, y_test_pred_forest)
f1_forest = f1_score(y_test, y_test_pred_forest)
```

In this case, classification report metrics are only slightly lower than for logistic regression, and predictions for class 0 are again a bit more accurate than for class 1. However, AUC score is a bit higher than for logistic regression:

In [66]:
```
auc_forest = roc_auc_score(y_test, prob_forest[:, 1])
auc_forest
```

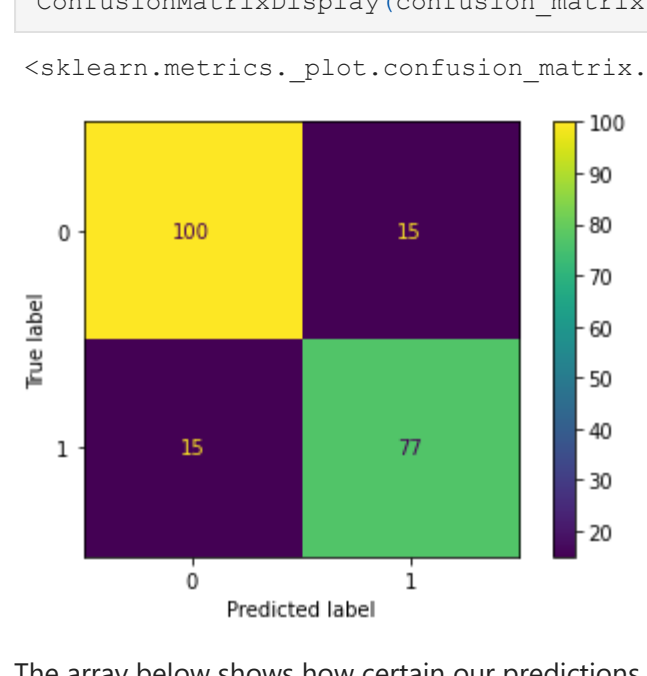
Out[66]: 0.9247637051039697

In [67]:
```
fpr_forest, tpr_forest, _ = roc_curve(y_test, prob_forest[:, 1])
precision_forest, recall_forest, _ = precision_recall_curve(y_test, prob_forest[:, 1])
```

Identical precision and recall values for both classes are explained by the same number of false positives and false negatives. But the matrix looks very similar to the logistic regression case:

In [68]:
```
ConfusionMatrixDisplay(confusion_matrix(y_test, y_test_pred_forest)).plot()
```

Out[68]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c98150de20>

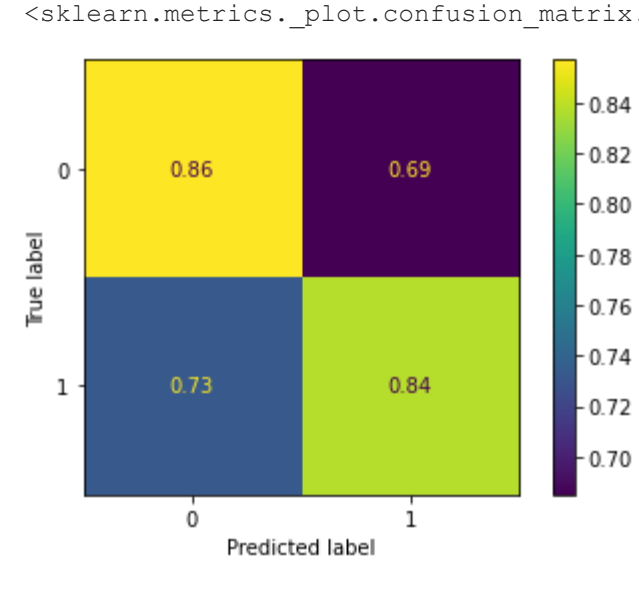


The array below shows how certain our predictions are for each confusion matrix entry. Similarly to the logistic regression model (and contrary to the decision tree), incorrect predictions are less certain on average:

In [69]:
```
prob_true_0_pred_1_forest = np.round(np.mean(np.max(prob_forest[(y_test==0) & (y_test_pred_forest==1), :], axis
prob_true_1_pred_1_forest = np.round(np.mean(np.max(prob_forest[(y_test==1) & (y_test_pred_forest==1), :], axis
prob_true_0_pred_0_forest = np.round(np.mean(np.max(prob_forest[(y_test==0) & (y_test_pred_forest==0), :], axis
prob_true_1_pred_0_forest = np.round(np.mean(np.max(prob_forest[(y_test==1) & (y_test_pred_forest==0), :], axis
```

In [70]:
```
prob_cm_forest = np.array([[prob_true_0_pred_0_forest, prob_true_0_pred_1_forest],
                           [prob_true_1_pred_0_forest, prob_true_1_pred_1_forest]])
ConfusionMatrixDisplay(prob_cm_forest).plot()
```

Out[70]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c984ac74c0>



Final Model Recommendation

After fitting three classification models, we will compare them and select the best one.

Classification report metrics (precision, recall, accuracy, F1-score) are very similar for logistic regression and random forest models (about 0.85). Confusion matrices also have almost the same counts.

The results for decision tree are a bit different. Its great advantage is interpretability - it is easy to visualize and understand, especially in our case when only one feature is selected. But contrary to other models, precision and recall values differ for the positive category by more than 0.1 (0.78 vs 0.9). Although average metric values are similar to other models, its AUC value is significantly lower (0.85 vs 0.92). These differences will be illustrated graphically below.

Also, we compared prediction probabilities for all three models. Logistic regression and random forest assign lower prediction probabilities for incorrect predictions (about 0.85 for correct vs 0.7 for incorrect). For the decision tree model however, the assigned probabilities depend only on the predicted label, regardless of whether the prediction is correct. So that is another drawback of the decision tree model.

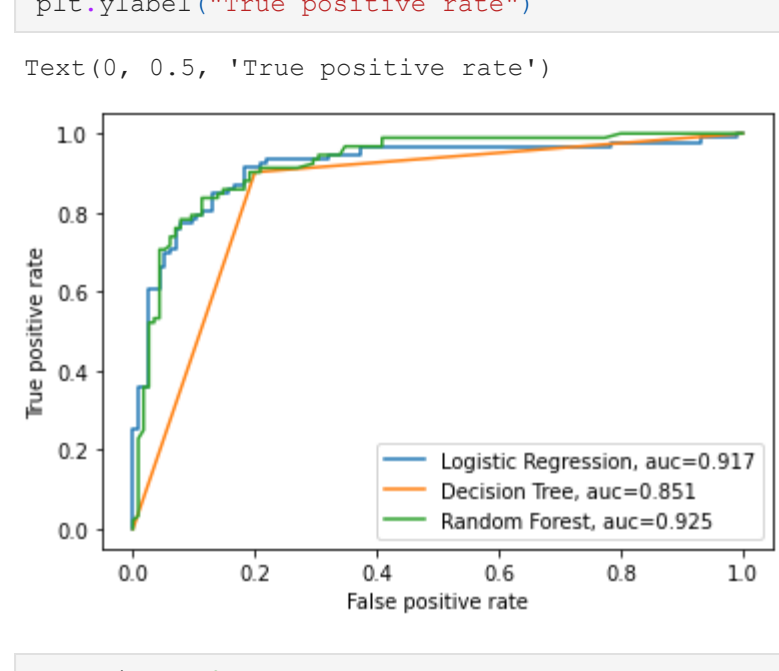
Below, ROC and precision-recall curves are plotted for all three models. Again, decision tree is doing the worst - its area under the curve is the lowest, since it has the only 2 thresholds (the same number as leaf nodes) for which to calculate FPR and TPR (the same for the precision-recall plot). Logistic regression and random forest are similar. AUC for the random forest model is slightly higher than for logistic regression. However, the precision-recall plot shows that random forest has worse precision rates for low recall values.

In [75]:
```
plt.figure(0).clf()

plt.plot(fpr_lr,tpr_lr,label="Logistic Regression, auc="+str(np.round(auc_lr, 3)))
plt.plot(fpr_tree,tpr_tree,label="Decision Tree, auc="+str(np.round(auc_tree, 3)))
plt.plot(fpr_forest,tpr_forest,label="Random Forest, auc="+str(np.round(auc_forest, 3)))

plt.legend(loc=0)
plt.xlabel("False positive rate")
plt.ylabel("True positive rate")
```

Out[75]: Text(0, 0.5, 'True positive rate')

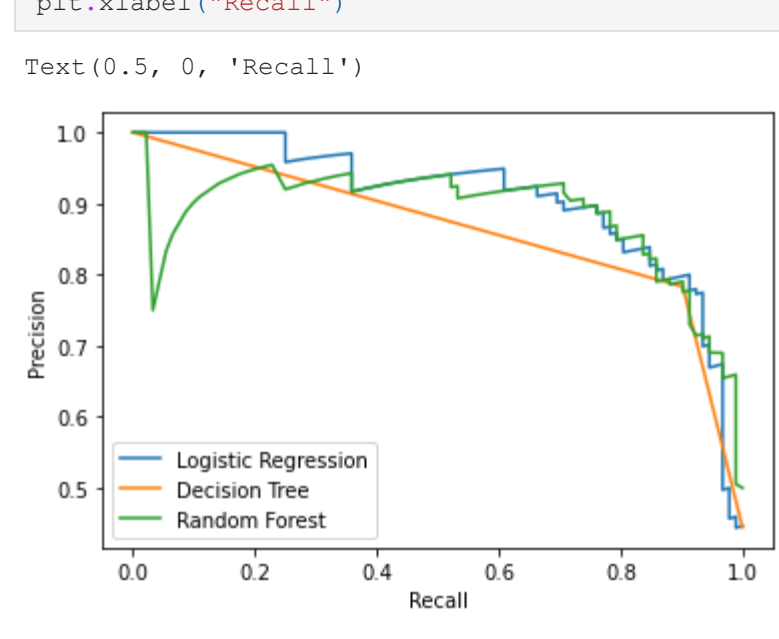


In [73]:
```
plt.figure(0).clf()

plt.plot(recall_lr,precision_lr,label="Logistic Regression")
plt.plot(recall_tree,precision_tree,label="Decision Tree")
plt.plot(recall_forest,precision_forest,label="Random Forest")

plt.legend(loc=0)
plt.ylabel("Precision")
plt.xlabel("Recall")
```

Out[73]: Text(0.5, 0, 'Recall')



In the previous part, we examined whether the logistic regression model adhers the necessary assumptions. We had doubts about it, so selecting the random forest model might be sensible, since it does not rely on any formal distributional assumptions.

So the final recommendation is to use the **random forest** model.

Key Findings And Insights

As we discussed above, random forest model is considered to be the best for this analysis. However, all three models give fairly good results in terms of prediction accuracy. Below is the summary of classification metrics for all the models (precision, recall and F1-score are given for class 1):

In [92]:
```
results_df=pd.DataFrame(data={'Logistic Regression': [acc_lr, prec_lr, rec_lr, f1_lr, auc_lr],
                              'Decision Tree': [acc_tree, prec_tree, rec_tree, f1_tree, auc_tree],
                               'Random Forest': [acc_forest, prec_forest, rec_forest, f1_forest, auc_forest],
                              'Metrics': ['Accuracy', 'Precision', 'Recall', 'F1 Score', 'AUC']})
results_df.set_index('Metrics')
```

Out[92]:

	Logistic Regression	Decision Tree	Random Forest
Metrics			
Accuracy	0.859903	0.845411	0.855072
Precision	0.838710	0.783019	0.836957
Recall	0.847826	0.902174	0.836957
F1 Score	0.843243	0.838384	0.836957
AUC	0.917013	0.851087	0.924764

Except precision for the decision tree model, all metrics exceed 0.8, and accuracy for logistic regression and random forest is above 0.85. Inconsistency of metric values for the decision tree model was discussed above and is most likely caused by excessive tree pruning.

The metrics tend to have lower values for class 1 cases across all our models. This could be explained by large values of outliers for class 1 that were observed in the data exploration part.

Apart from predictability, let's analyze which variables are found to be the most important in our models.

The most important variable in all three models is A8_1. For logistic regression and random forest, it is several times more important than other features. Also, it is the only variable in the decision tree model, and is enough for a quite good prediction accuracy.

The other similarity between the logistic regression and random forest models is a relatively high importance of numeric variables - A13, A14, A7 and A10 are common for both models. Other numeric variables are more important for the random forest model, while the logistic regression gives more importance to some dummy variables (A4_2, A5_9, A6_8, A5_14). On the other hand, some of these variables have high VIF values (see the analysis above), and thus might be useful to remove from the dataset.

So the analysis of variable importance justifies the selection of random forest for this study.

Suggestions

The initial analysis shows that the numeric features had some outliers that we did not remove, because we were unaware of their nature. Examining them is much harder due to the missing feature names. A deeper analysis of potential outliers could be useful to decide whether to preserve them in the dataset. Model predictability might improve upon their removal, especially of logistic regression, since it relies on linearity assumption.

Although entry numbers for both classes are similar (55% and 45%), we could apply oversampling and/or undersampling techniques to make the dataset perfectly balanced. These approaches would have their own disadvantages, such as information loss or a risk of overfitting.

Feature multicollinearity analysis after fitting logistic regression showed that some (mostly categorical) variables have high variance inflation factors. Since information provided by them is redundant, re-fitting without them might be another way to improve our models.

Also, some other classification models could be experimented with. We used random forest as an **averaging** method, based on many independent trees. **Boosting** methods (such as AdaBoost) rely on building estimators sequentially, increasing weights on samples that were previously predicted incorrectly.

In [ ]: