notebook

May 20, 2020

IB031 Project

- Source
- Testing csv
- Training csv

In this project we will implement several models to predict credibility of a loan applicant. We will use the Loan Prediction Problem Dataset from Kaggle. The structure of the project is as follows:

1. Section **??**
2. Section **??**
3. Section **??**
4. Section **??**
5. Section **??**
6. Section **??**
7. Section **??**
8. Section **??**
9. Section **??**

0.1 1. Exploratory Analysis

The dataset consists of basic information about applicants. There are 13 columns with about 1000 rows. First of all, we will deal with the data types.

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

sns.set()
```

Since pandas loads categorical data types as *'object'* dtype by default, we want to convert them back to category after loading the csv. We are also dropping the "Loan_Status" column, which will later provide labels for classification. Unfortunately, this particular dataset does not contain a set of testing labels, therefore we were forced to split an already sparse training set into 2 parts.

[4]:
```
from sklearn.model_selection import train_test_split

#X_test = pd.read_csv("./test_Y3wMUE5_7gLdaTN.csv", index_col="Loan_ID")
```

```
dataset = pd.read_csv("./train_u6lujuX_CVtuZ9i.csv", index_col="Loan_ID")

for col in dataset.columns:
    if dataset[col].dtype == 'object':
        dataset[col] = dataset[col].astype('category')

X, y = dataset.drop(["Loan_Status"], axis=1), dataset["Loan_Status"][:]
y = y.map({"Y": True, "N": False})

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2)
```

[5]:
```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 614 entries, LP001002 to LP002990
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             601 non-null    category
 1   Married            611 non-null    category
 2   Dependents         599 non-null    category
 3   Education          614 non-null    category
 4   Self_Employed      582 non-null    category
 5   ApplicantIncome    614 non-null    int64
 6   CoapplicantIncome  614 non-null    float64
 7   LoanAmount         592 non-null    float64
 8   Loan_Amount_Term   600 non-null    float64
 9   Credit_History     564 non-null    float64
 10  Property_Area      614 non-null    category
dtypes: category(6), float64(4), int64(1)
memory usage: 33.0+ KB
```

We have dropped the 'Loan_Status' column, which will provide labels during classification

[6]:
```
X.shape
```

[6]: (614, 11)

[7]:
```
y.shape
```

[7]: (614,)

The dataset had been split into training and testing subsets in around 1.67:1 ratio.

We can look at the amount of distinct values of selected features.

[8]:
```
# value count of just some selected columns

print(X.Gender.value_counts()) # oh yeah boi
```

```
print(X.Married.value_counts())
print(X.Education.value_counts())
print(X.Property_Area.value_counts())
```

```
Male      489
Female    112
Name: Gender, dtype: int64
Yes    398
No     213
Name: Married, dtype: int64
Graduate        480
Not Graduate    134
Name: Education, dtype: int64
Semiurban    233
Urban        202
Rural        179
Name: Property_Area, dtype: int64
```

[9]:
```
X.head()
```

[9]:
```
         Gender Married Dependents     Education Self_Employed  \
Loan_ID
LP001002   Male      No          0      Graduate            No
LP001003   Male     Yes          1      Graduate            No
LP001005   Male     Yes          0      Graduate           Yes
LP001006   Male     Yes          0  Not Graduate            No
LP001008   Male      No          0      Graduate            No

          ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
Loan_ID
LP001002             5849                0.0         NaN             360.0
LP001003             4583             1508.0       128.0             360.0
LP001005             3000                0.0        66.0             360.0
LP001006             2583             2358.0       120.0             360.0
LP001008             6000                0.0       141.0             360.0

          Credit_History Property_Area
Loan_ID
LP001002             1.0         Urban
LP001003             1.0         Rural
LP001005             1.0         Urban
LP001006             1.0         Urban
LP001008             1.0         Urban
```

[10]:
```
X.isnull().sum()
```

[10]:
```
Gender                13
Married                3
Dependents            15
Education              0
Self_Employed         32
ApplicantIncome        0
CoapplicantIncome      0
LoanAmount            22
Loan_Amount_Term      14
Credit_History        50
Property_Area          0
dtype: int64
```

As we can see, there are a lot of null values which will need to be imputed. Since there is no column with more missing data than present, we do not need to drop any.

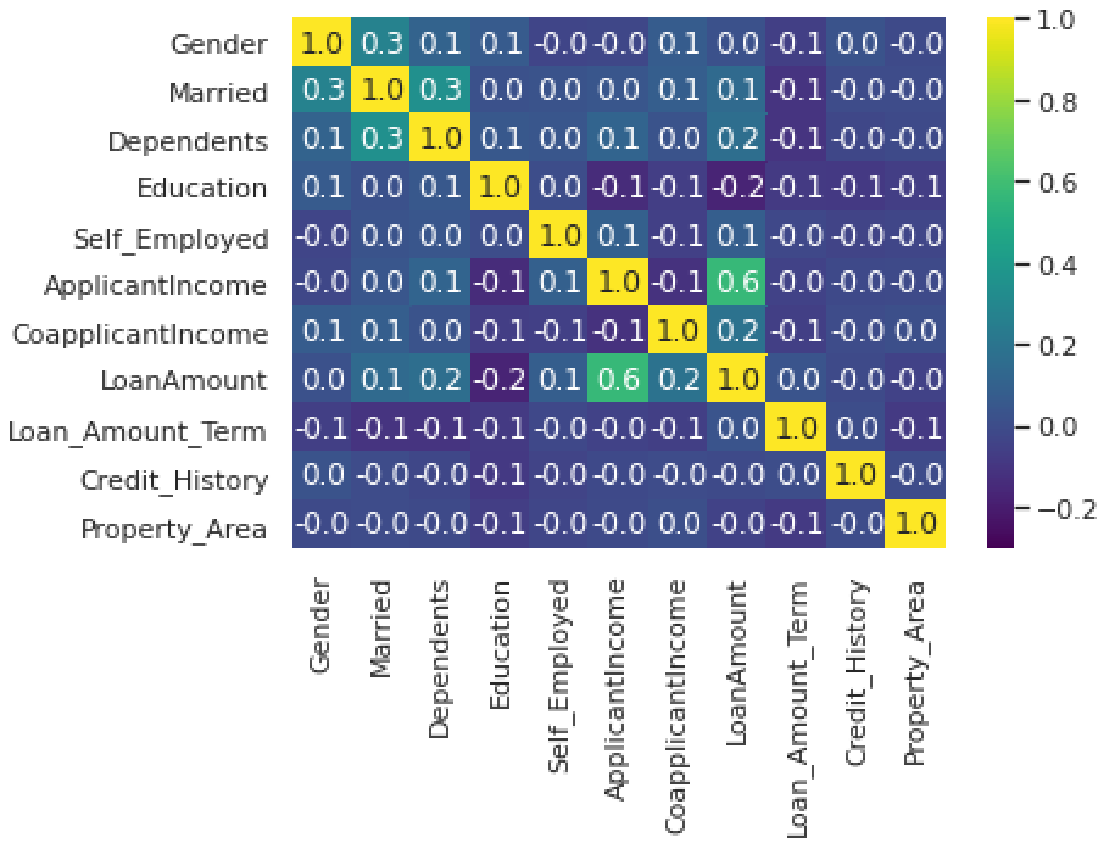
[11]:
```
X.describe()
```

[11]:
```
       ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
count       614.000000         614.000000  592.000000         600.00000
mean       5403.459283        1621.245798  146.412162         342.00000
std        6109.041673        2926.248369   85.587325          65.12041
min         150.000000           0.000000    9.000000          12.00000
25%        2877.500000           0.000000  100.000000         360.00000
50%        3812.500000        1188.500000  128.000000         360.00000
75%        5795.000000        2297.250000  168.000000         360.00000
max       81000.000000       41667.000000  700.000000         480.00000

       Credit_History
count      564.000000
mean         0.842199
std          0.364878
min          0.000000
25%          1.000000
50%          1.000000
75%          1.000000
max          1.000000
```

Here we have some essential statistics about our dataset. For instance, we can see, that the most frequent length of loan term is about 1 year, with maximum being 1.5 year and minimum just 12 days. Furthermore, coapplicants have much lower income than applicants.

[62]:
```
corr_X = X[:]
for col in {"Property_Area", "Dependents", "Gender", "Married", "Education",␣
 ↪"Self_Employed"}:
    corr_X[col] = corr_X[col].cat.codes

sns.heatmap(corr_X.corr(), annot=True, vmin=-0.3, cmap="viridis", fmt=".1f")
```

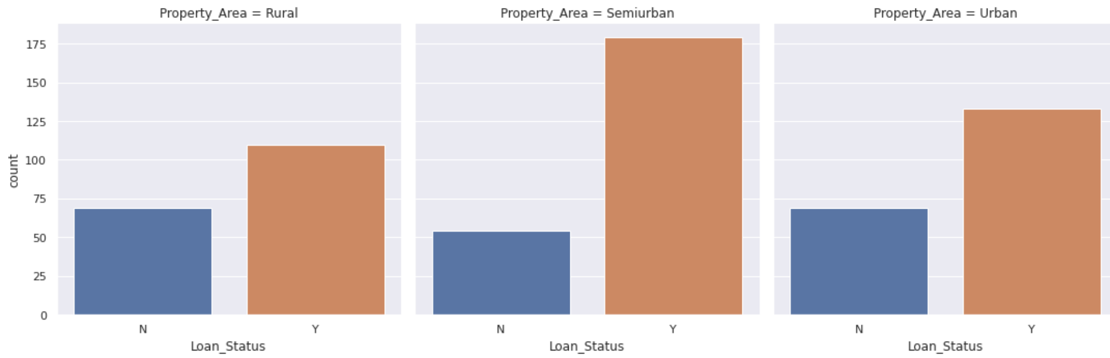
```
plt.show()
```

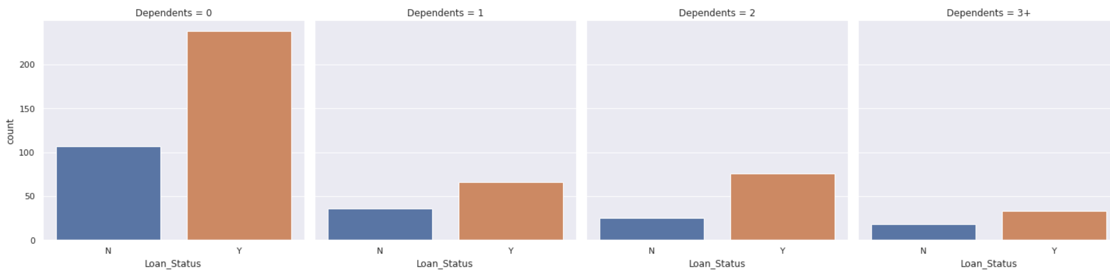


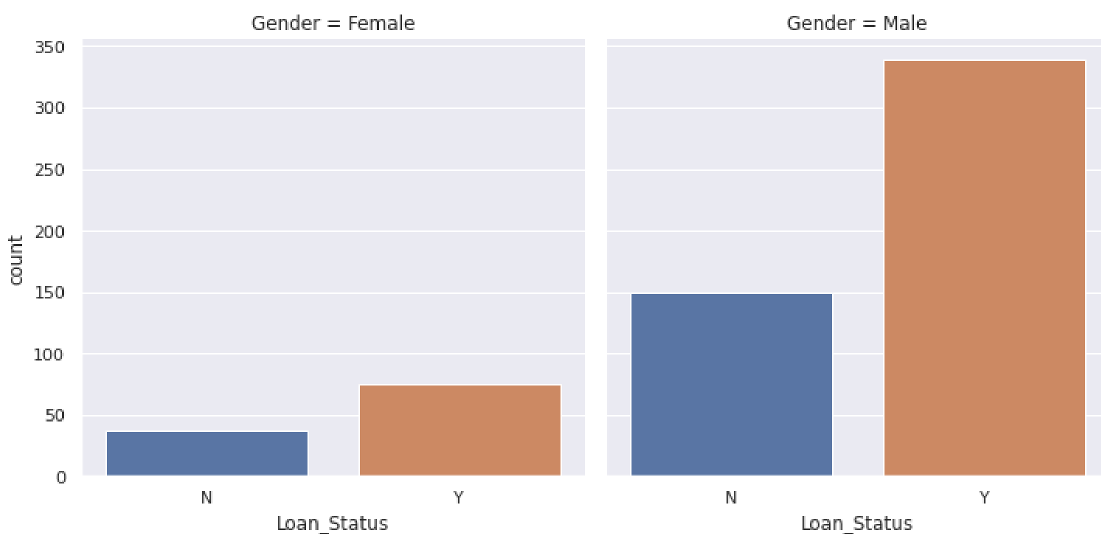
We also computed and plotted correlation matrix of features in the dataset. There is only one pair of features, loan amount and applicant income, which can be considered to be mildly positively correlated. Other features, have nearly no correlation among them. This results is quite suprising, as one would expect higher correlation of features.

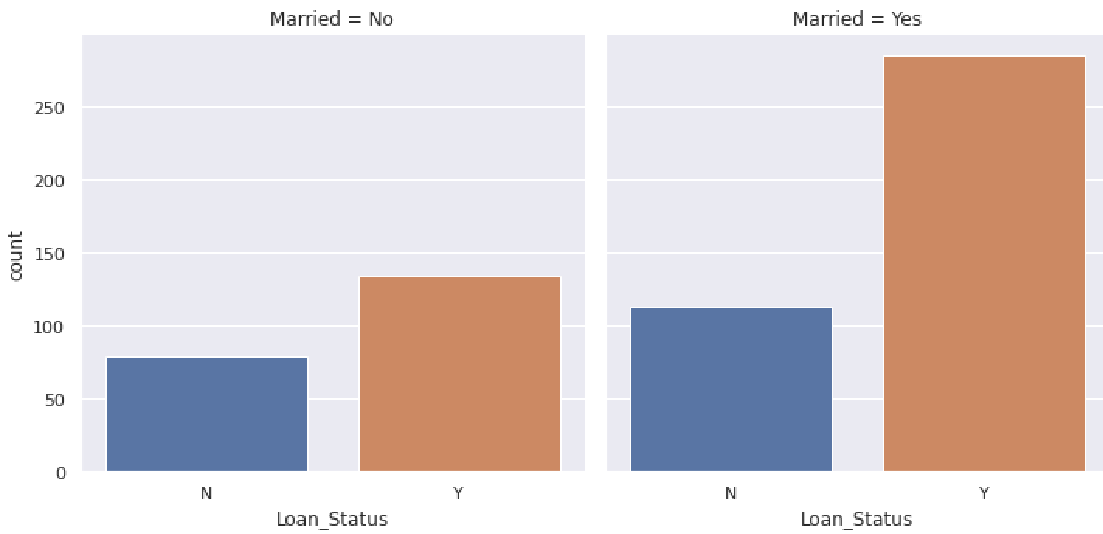
[13]:
```
sns.catplot("Loan_Status", col="Property_Area", data=dataset, kind="count")
sns.catplot("Loan_Status", col="Dependents", data=dataset, kind="count")
sns.catplot("Loan_Status", col="Gender", data=dataset, kind="count")
sns.catplot("Loan_Status", col="Married", data=dataset, kind="count")
sns.catplot("Loan_Status", col="Education", data=dataset, kind="count")
```

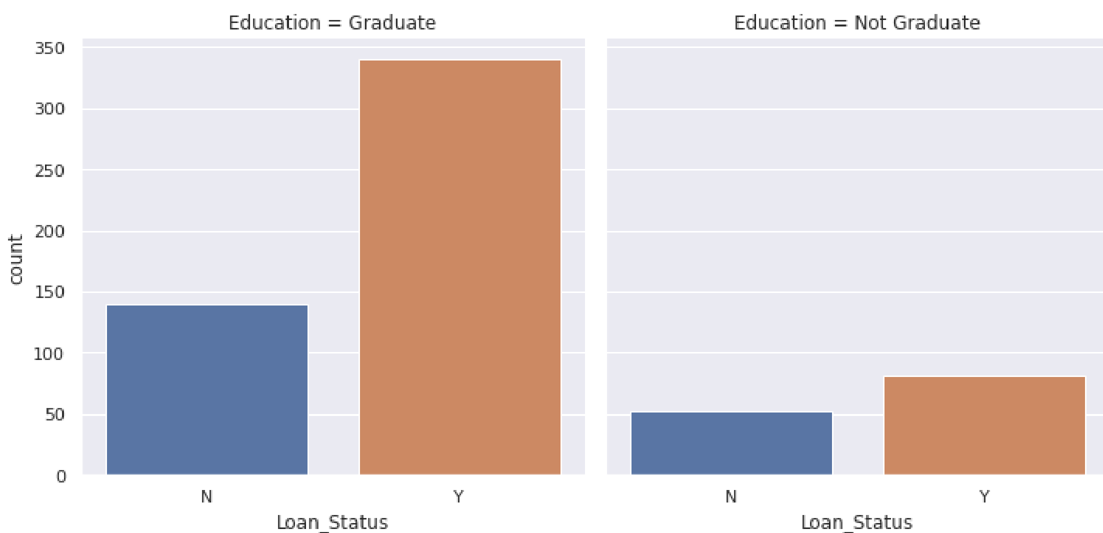
[13]: <seaborn.axisgrid.FacetGrid at 0x7fc01d075978>


Property_Area = Rural
Property_Area = Semiurban
Property_Area = Urban
count
Loan_Status
N
Y

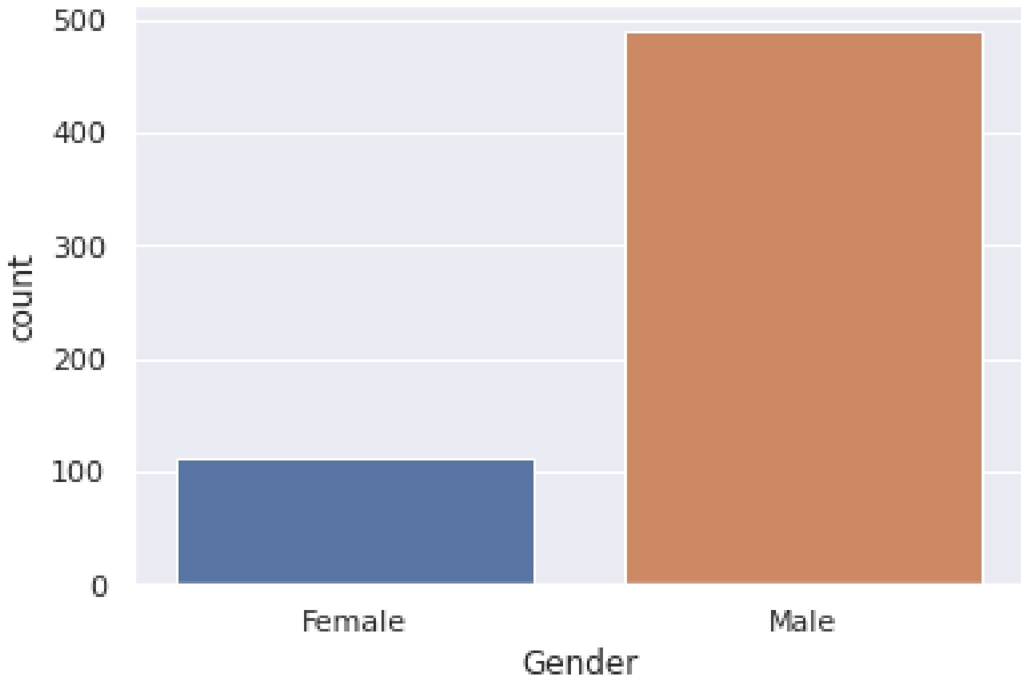

Dependents = 0
Dependents = 1
Dependents = 2
Dependents = 3+
count
Loan_Status
N
Y

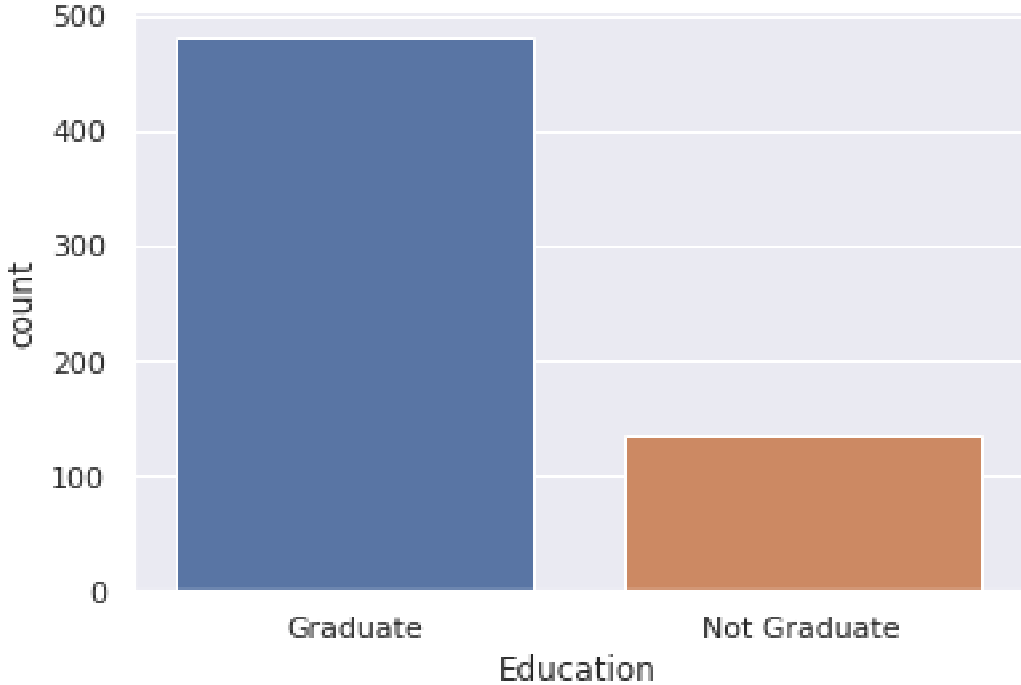

Gender = Female
Gender = Male
count
Loan_Status
N
Y

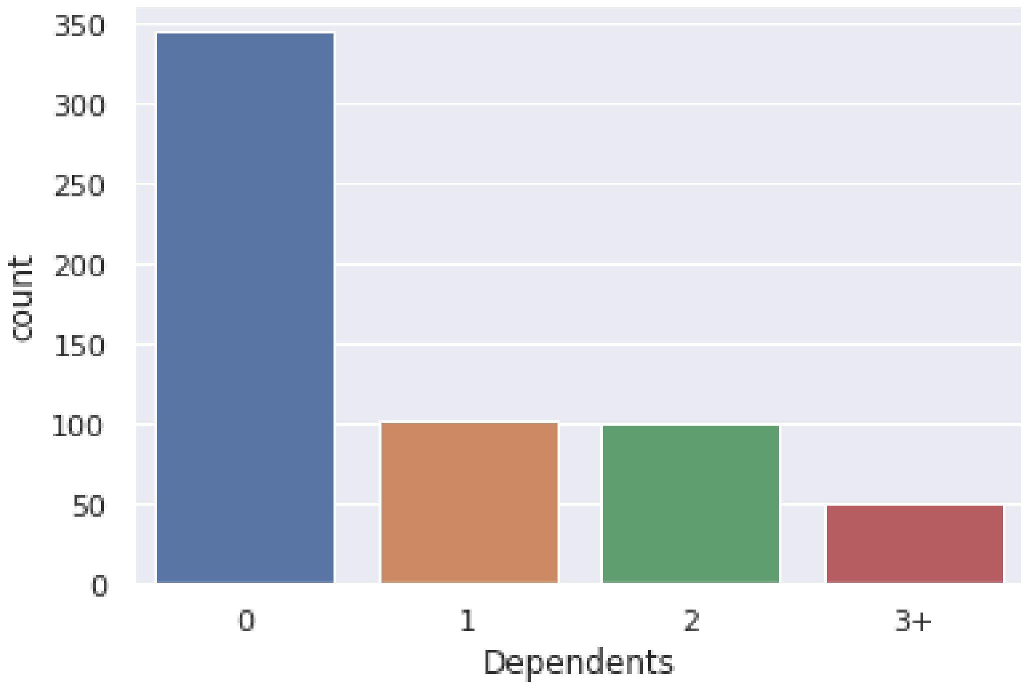


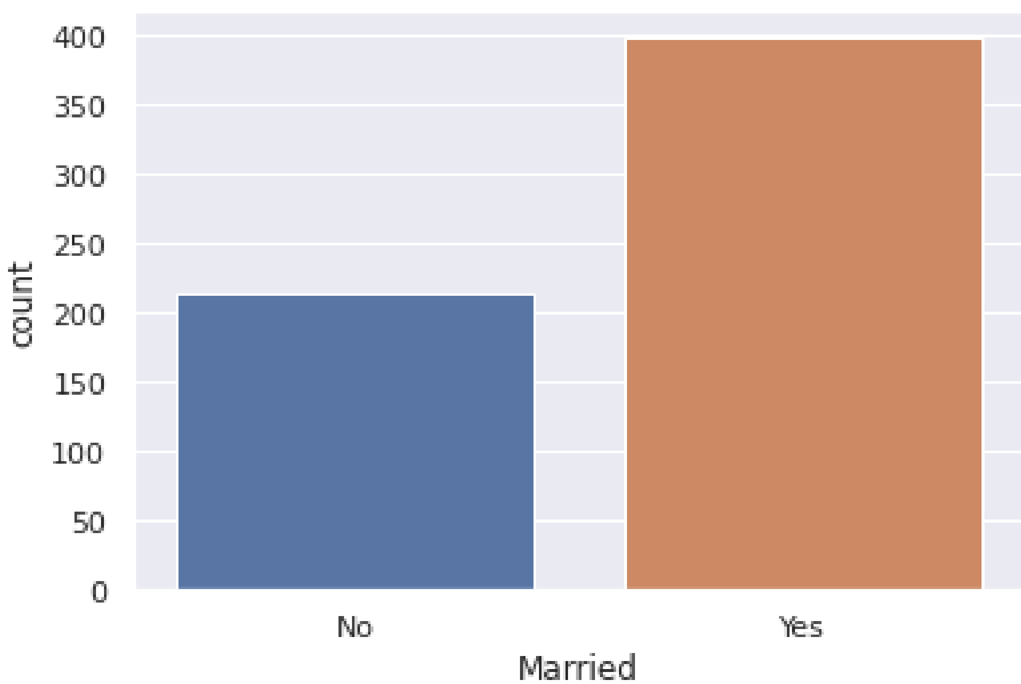


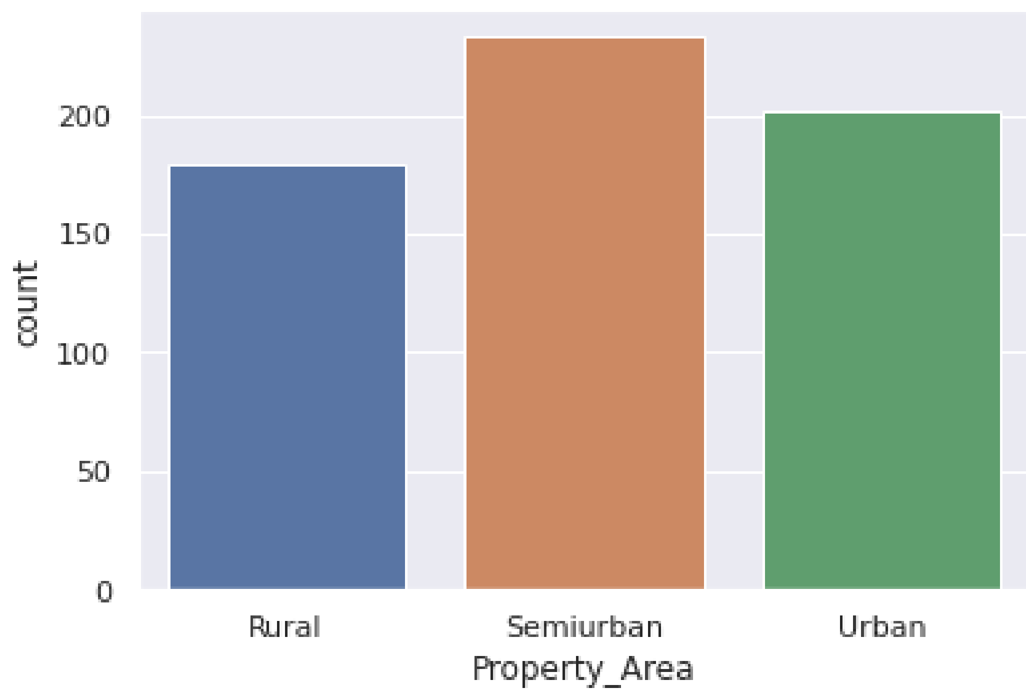
[67]:
```
for col in {"Property_Area", "Dependents", "Gender", "Married", "Education",␣
↪"Self_Employed"}:
    sns.countplot(X[col])
    plt.show()

for col in {"ApplicantIncome", "CoapplicantIncome", "Credit_History",␣
↪"LoanAmount", "Loan_Amount_Term"}:
    sns.distplot(X[col])
```

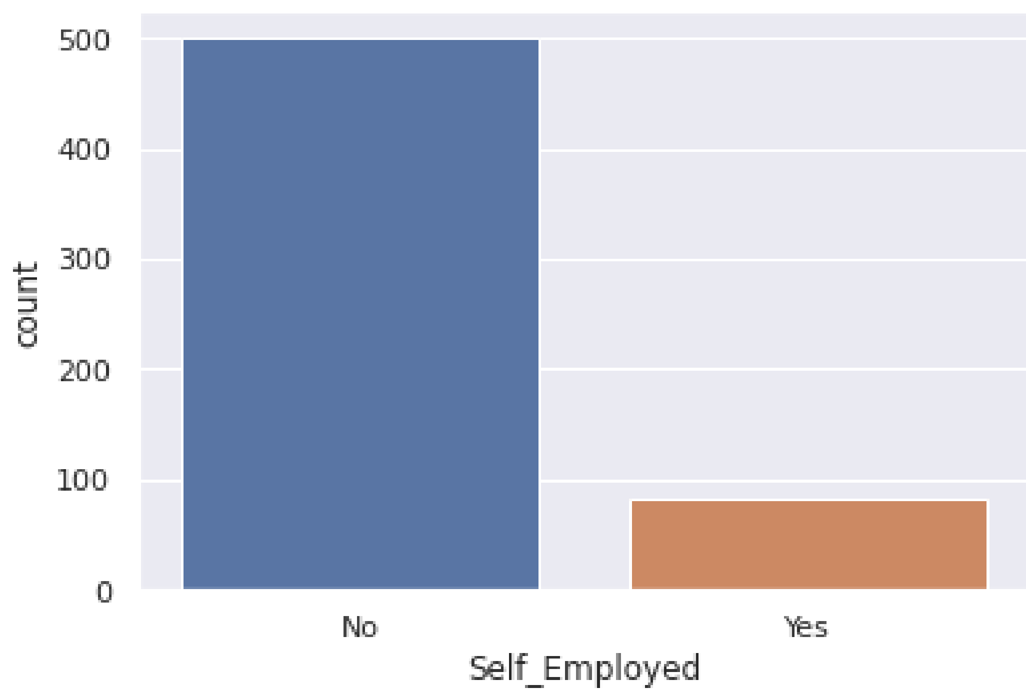

500
400
300
200
100
0
count
Female
Male
Gender

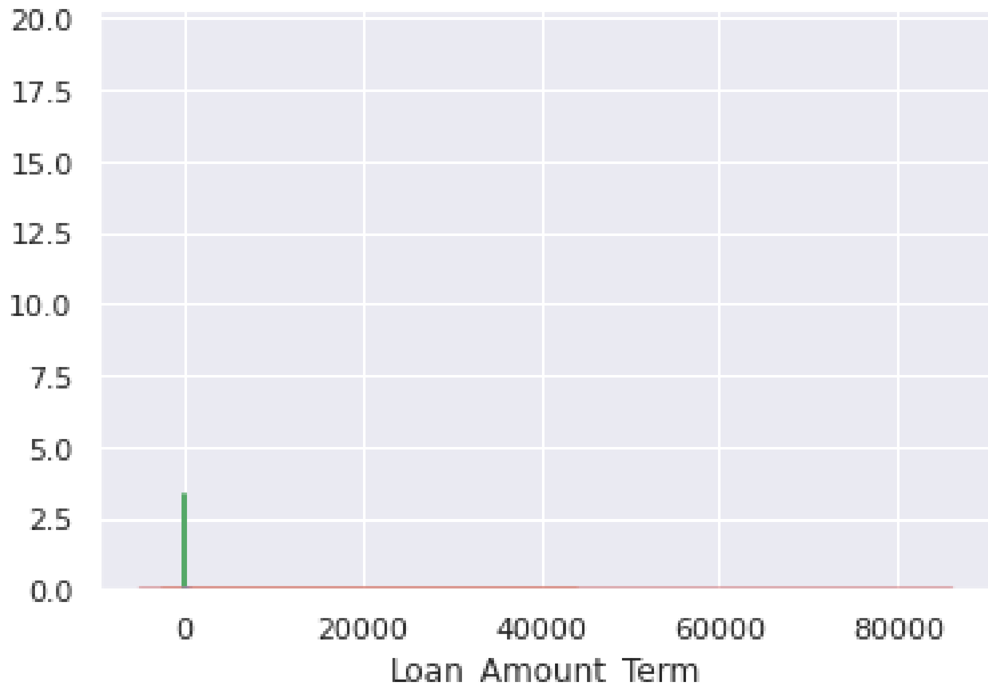

500
400
300
200
100
0
count
Graduate
Not Graduate
Education


350
300
250
200
150
100
50
0
count
0
1
2
3+
Dependents


400
350
300
250
200
150
100
50
0
count
No
Yes
Married


200
150
100
50
0
count
Rural
Semiurban
Urban
Property_Area

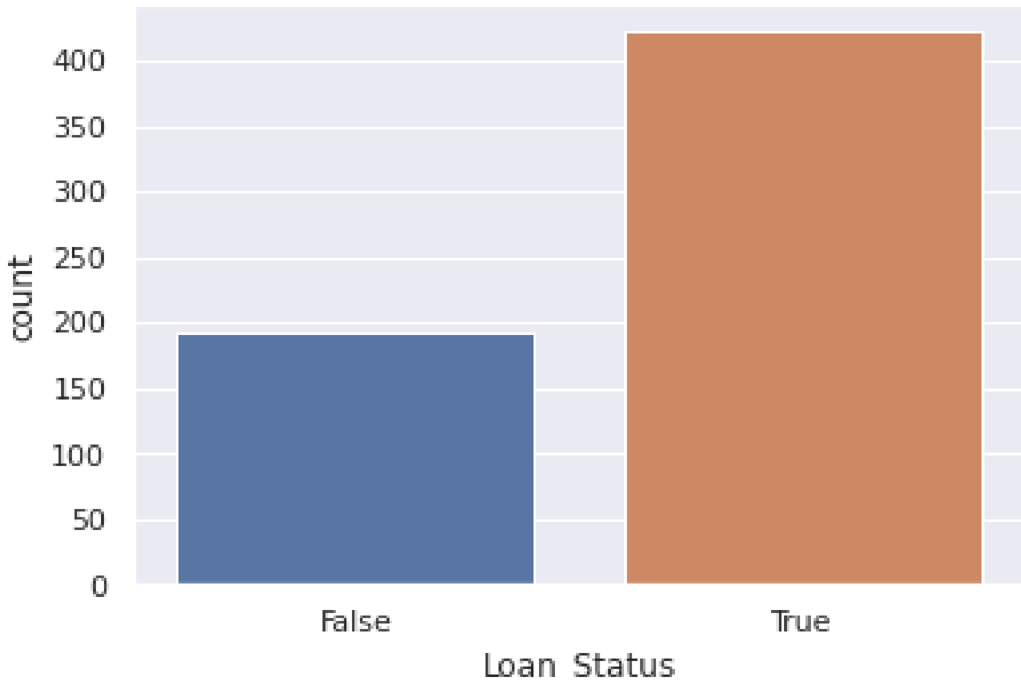

500
400
300
200
100
0
count
No
Yes
Self_Employed



Most of the features are scattered along the x-axis. The only exception is Loan Amount, which roughly follows the Chi-squared distribution.

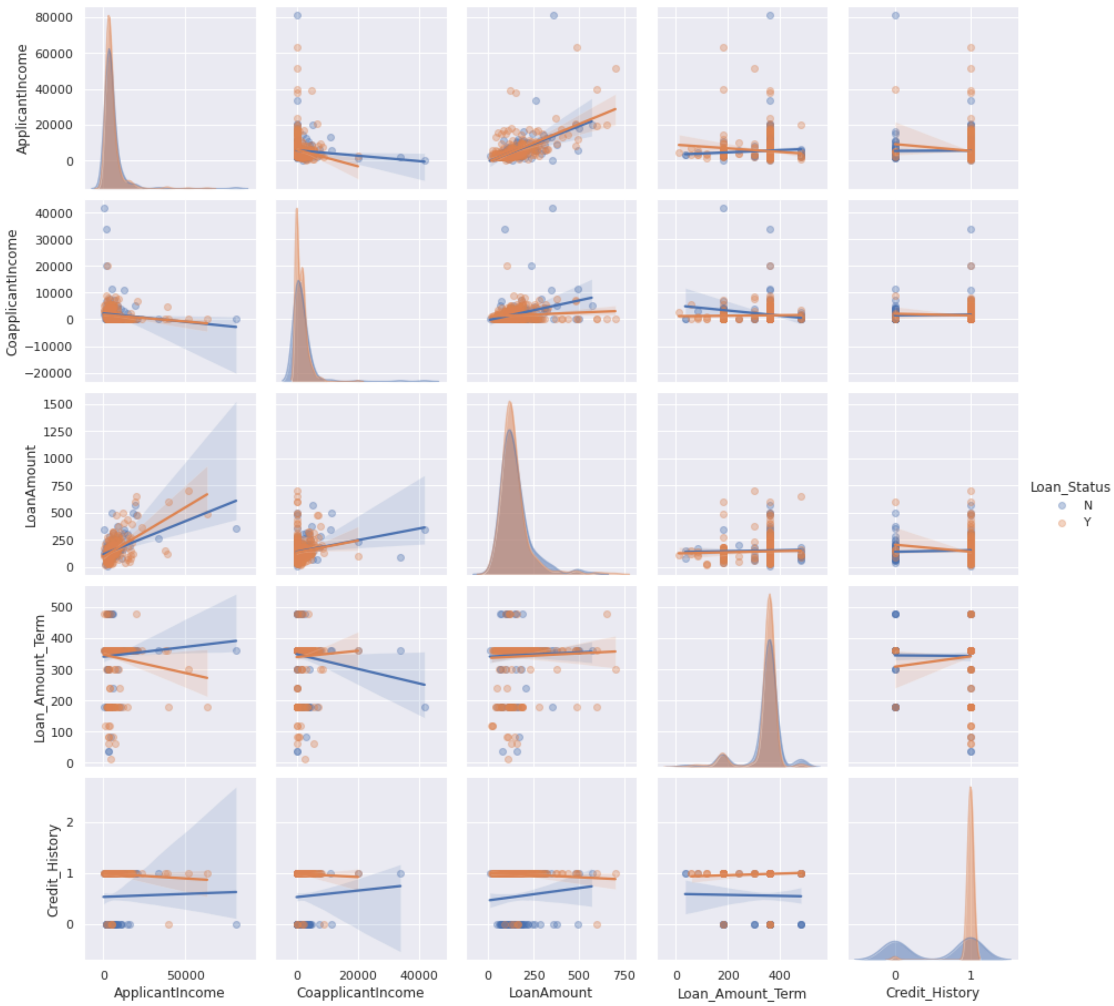
[15]:
```
sns.countplot(y)
```

[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc01cc9c630>



[65]:
```
sns.pairplot(dataset, hue="Loan_Status", kind="reg", diag_kws={"alpha": 0.5},␣
 ↪plot_kws={"scatter_kws": {"alpha": 0.35}})
```

[65]: <seaborn.axisgrid.PairGrid at 0x7fbf6a0a8cc0>



Credit history seems to have the highest impact on deciding whether the loan should be approved or not. Clients with low credit score seem more likely to be denied.

There are some cases where ApplicantIncome and CoapplicantIncome are very low, their loan request gets approved, whereas in a few cases the CoapplicantIncome is very high and ApplicantIncome is very low, the request gets denied.

0.2 2. Data preprocessing

[17]:
```
y.isna().sum()
```

[17]: 0

Good, there are no N/A labels. We do not need to drop any rows. Besides that, dropping a significant part of the dataset could have misrepresenting effects.

Firstly, we will impute the missing values with `SimpleImputer` and strategy `most_frequent`. As `SimpleImputer` returns an array, we will transform it back to a pandas dataframe. Another step is encoding the categorical features using `OneHotEncoder`. Lastly, we will scale the features with `StandardScaler`. Their mean will therefore be 0 and variance 1.

The labels need significantly less preprocessing. Encoding their boolean values is sufficient.

[18]:
```
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder, StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.impute import SimpleImputer

class NpToDf:
    columns = []

    def __init__(self, columns=None):
        self.columns = columns

    def fit(self, X, *args, **kwargs):
        return X

    def fit_transform(self, X, *args, **kwargs):
        return self.transform(X)

    def transform(self, X, *args, **kwargs):
        return pd.DataFrame(data=X, columns=self.columns)

pipe_X = make_pipeline(
    SimpleImputer(missing_values=np.nan, strategy="most_frequent"),
    NpToDf(X_train.columns),
    make_column_transformer(
        (OneHotEncoder(), ['Gender', 'Married', 'Education', 'Self_Employed',␣
↪'Property_Area', 'Dependents']),
        remainder=StandardScaler()),
    NpToDf(),
)

pipe_y = make_pipeline(
    OrdinalEncoder()
)
```

[19]:
```
pipe_X.fit_transform(X_train)
train_X = pipe_X.transform(X_train)
test_X = pipe_X.transform(X_test)

train_y = pipe_y.fit_transform(y_train.to_numpy().reshape(-1, 1)).reshape(-1)
test_y = pipe_y.fit_transform(y_test.to_numpy().reshape(-1, 1)).reshape(-1)
```

Some helper functions for training and evaluation of our models.

[20]:
```
from sklearn.metrics import mean_squared_error, f1_score
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt

def evaluate(clf, X_test, y_test):
    y_pred = clf.predict(X_test)
    scores = cross_val_score(clf, X_test, y_test, cv=10)

    print(f"RMSE: {mean_squared_error(y_test, y_pred, squared = False):.4f}")
    print(f"Accuracy: {scores.mean():.3f} ± {scores.std() * 2:.3f}")
    print("F1 Score: %.2f" % f1_score(y_test, y_pred, average='weighted'))
```

[21]:
```
from sklearn.metrics import plot_roc_curve

def roc(clf, test_X, test_y):
    plot_roc_curve(clf, test_X, test_y)
    plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Chance',␣
 ↪alpha=.8)
    plt.legend()
```

[22]:
```
from sklearn.metrics import plot_precision_recall_curve as prc
from sklearn.metrics import precision_recall_curve
```

[23]:
```
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import GridSearchCV

def confusion(clf, X_test, y_test):
    plot_confusion_matrix(clf, X_test, y_test, cmap=plt.cm.Blues)
```

[24]:
```
def get_gscv(clf, param_grid, verbose=1, **kwargs):
    gs = GridSearchCV(clf, param_grid, verbose=verbose, cv=kwargs.get("cv", 3),␣
 ↪n_jobs=kwargs.get("workers", -2))
    gs.fit(train_X, train_y, **kwargs)
    score = gs.score(test_X, test_y)
    print(f"Best parameters: {gs.best_params_}, with F1 score of {score:.2f}")
    return gs.best_estimator_
```

In the following 4 sections, we will always start by running a grid search tuning hyperparameters of our models. There is a wrapper for keras sequential models, which we will use for grid searching. All models will be trained on the same training set. We chose these evaluation metrics - **RMSE** - measures error of the predictions compared to actual values, the lower the better. - **Acurracy** computed with cross validation score - is the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined, the higher the better - **F1 score** - is weighted average of precision and recall, the best value is 1, the worst is 0 - **Receiver Operating Characteristic curve** - measures the ability of a model to distinguish between classes - **Precision**

Recall curve - shows the tradeoff between precision and recall for different threshold - **Confusion matrix** - shows the number of True Positive (TP), False Negative (FN), True Negative (TN), False Positive (FP) classifications.

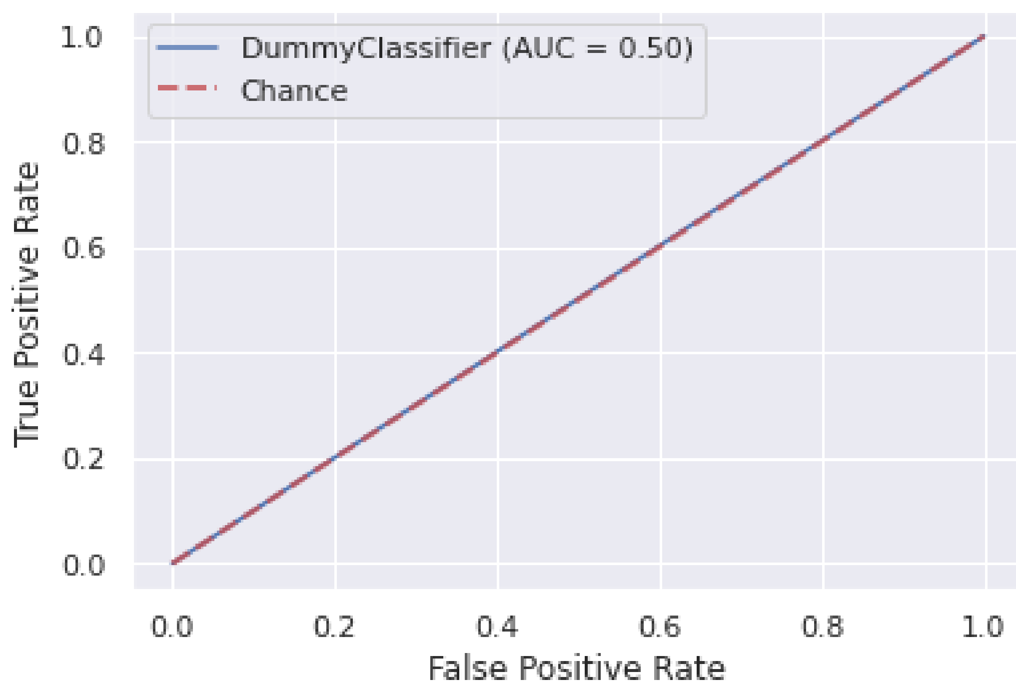
0.3 3. Naive baseline model

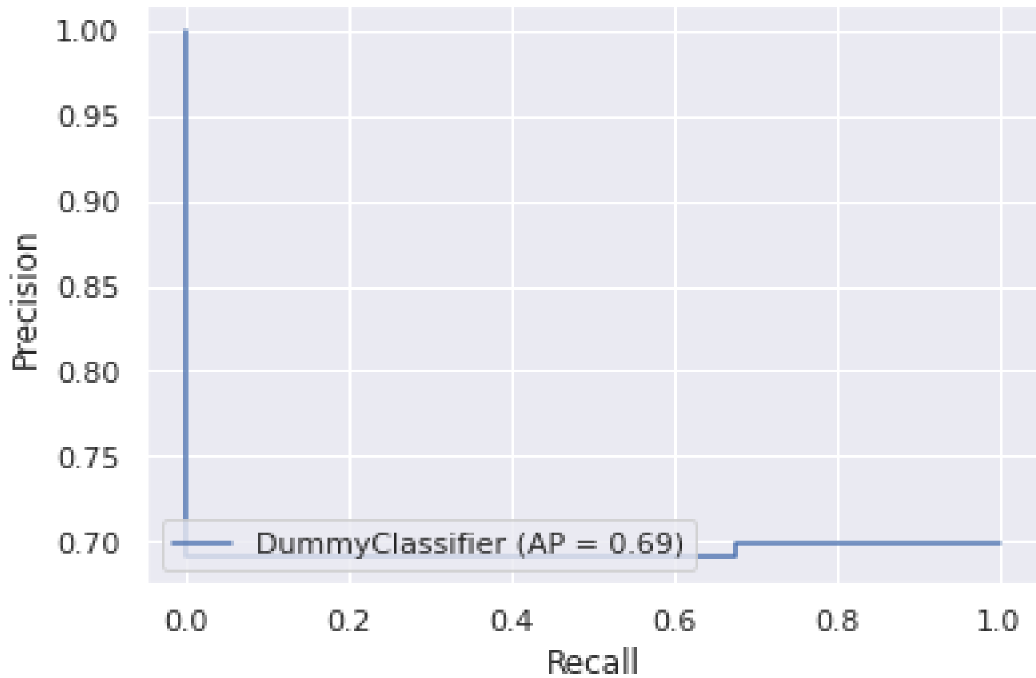
This is a simple classifier, which chooses the class based on training set class distribution.

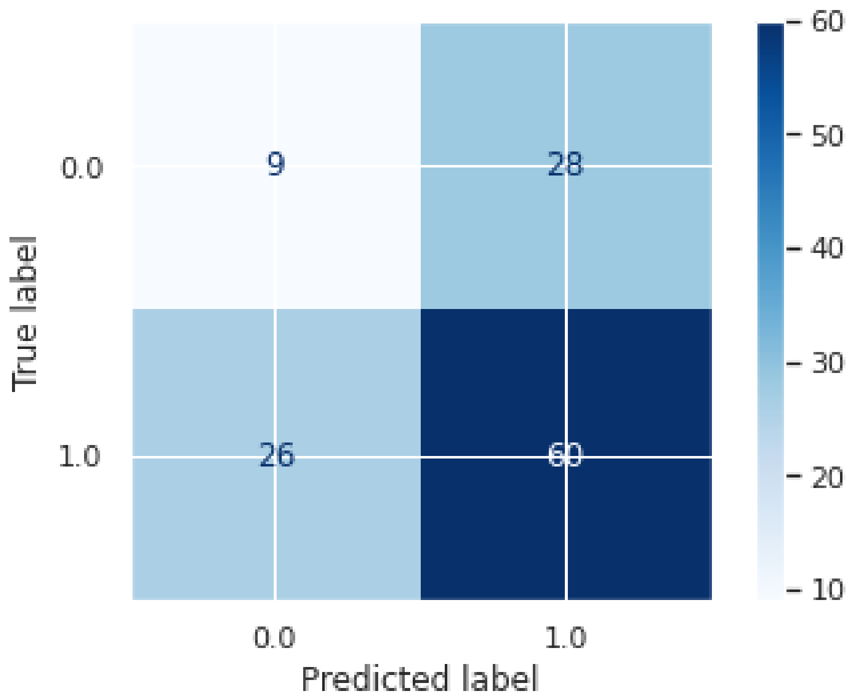
It is very basic and is affected by the chosen train/test split a lot.

[26]:
```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier(strategy="stratified")

dummy_clf.fit(train_X, train_y)

evaluate(dummy_clf, test_X, test_y)
roc(dummy_clf, test_X, test_y)
prc(dummy_clf, test_X, test_y)
confusion(dummy_clf, test_X, test_y)
plt.show()
```

```
RMSE: 0.6182
Accuracy: 0.577 ± 0.179
F1 Score: 0.61
```




1.00
0.95
0.90
0.85
0.80
0.75
0.70
Precision
DummyClassifier (AP = 0.69)
0.0
0.2
0.4
0.6
0.8
1.0
Recall


9
28
26
60
0.0
1.0
True label
0.0
1.0
Predicted label
60
50
40
30
20
10

0.4 4. Decision tree classifier

Decision trees are among the most used classification models. They iteratively split the dataset, until a tree conforming to given parameters has been constructed. In leaves they contain class labels. Internal nodes represent kind of a boolean test, usually a value of a sample's feature, according to which the algorithm chooses the respective edge on the way to leaves. The tests can also use entropy and information gain to choose the best edge. There are many to ways to construct a tree, therefore extensive hyperparameter tunning is suitable. Decision trees can also be pruned, etheir during construction of after it.

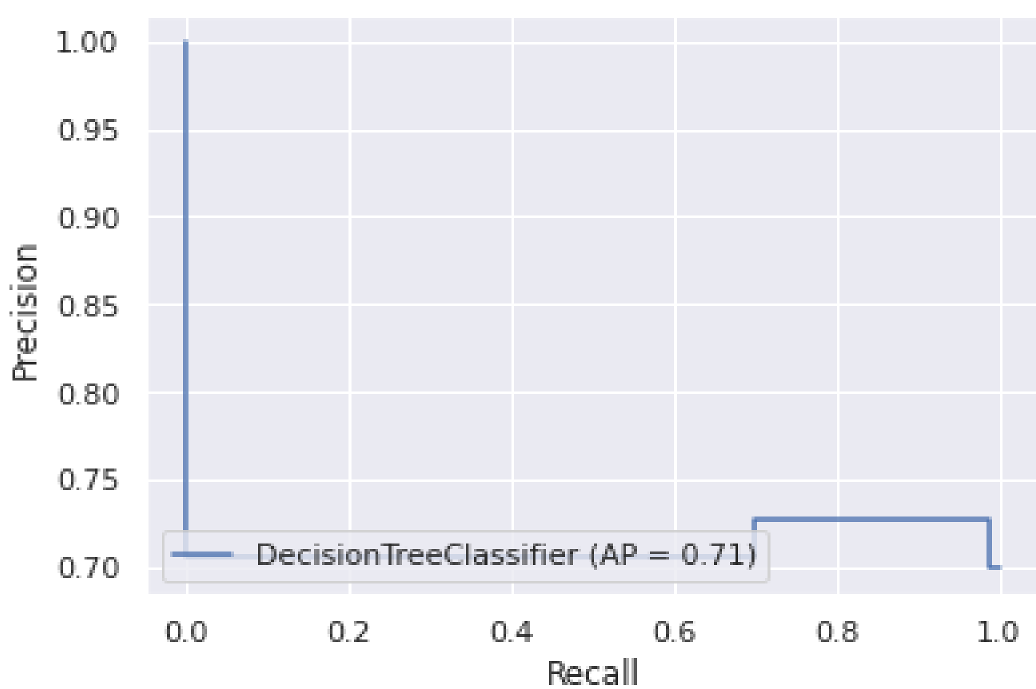
[27]:
```python
from sklearn.tree import DecisionTreeClassifier

dtree_clf = DecisionTreeClassifier()
dtree_values = {"criterion": ["gini", "entropy"],
                "max_depth": [1, 2, 5, 10, 16, None],
                "max_leaf_nodes": [6, 8, 10, 12, 20, None],
                "max_features": ["auto", "sqrt", "log2", 2, 4, 8]}

tree_clf = get_gscv(dtree_clf, dtree_values)
```

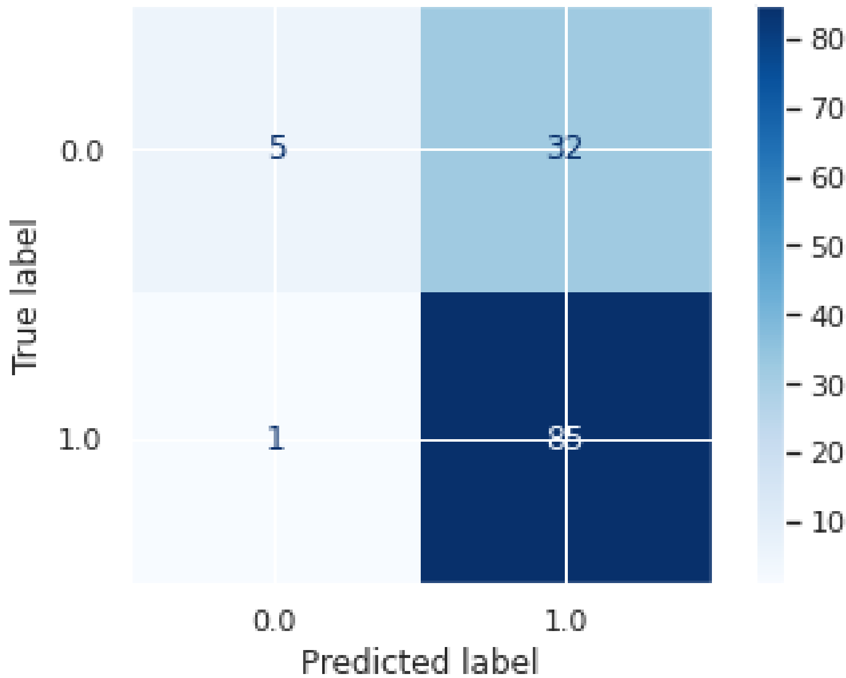
```
Fitting 3 folds for each of 432 candidates, totalling 1296 fits
[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=-2)]: Done 140 tasks      | elapsed:    2.7s
Best parameters: {'criterion': 'entropy', 'max_depth': 2, 'max_features': 8,
'max_leaf_nodes': 20}, with F1 score of 0.73
[Parallel(n_jobs=-2)]: Done 1296 out of 1296 | elapsed:    6.9s finished
```

[28]:
```python
evaluate(tree_clf, test_X, test_y)
roc(tree_clf, test_X, test_y)
prc(tree_clf, test_X, test_y)
confusion(tree_clf, test_X, test_y)
plt.show()
```

```
RMSE: 0.5180
Accuracy: 0.749 ± 0.245
F1 Score: 0.66
```


DecisionTreeClassifier (AUC = 0.53)
Chance
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


DecisionTreeClassifier (AP = 0.71)
Precision
Recall
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0.0
0.2
0.4
0.6
0.8
1.0



0.5 5. KNN classifier

Another popular classification algorithm, an example of instace-based learning or lazy learning. This time, all distances from a data point to other points are computed, and k-closest neighbours are chosen. Then, the class memberships of the *k-closest* members are considered, with the original data point taking a class label from the most occuring one among its *k-closest* neighbours. For the distance metrics, *Euclidean* or *Hamming* distances are usually used. There is a tradeoff in the number of *k-closest* neighbours. Smaller *k*, signifies the result of noise on classification, but makes the various classses more distinct and vice versa with higher *k*.

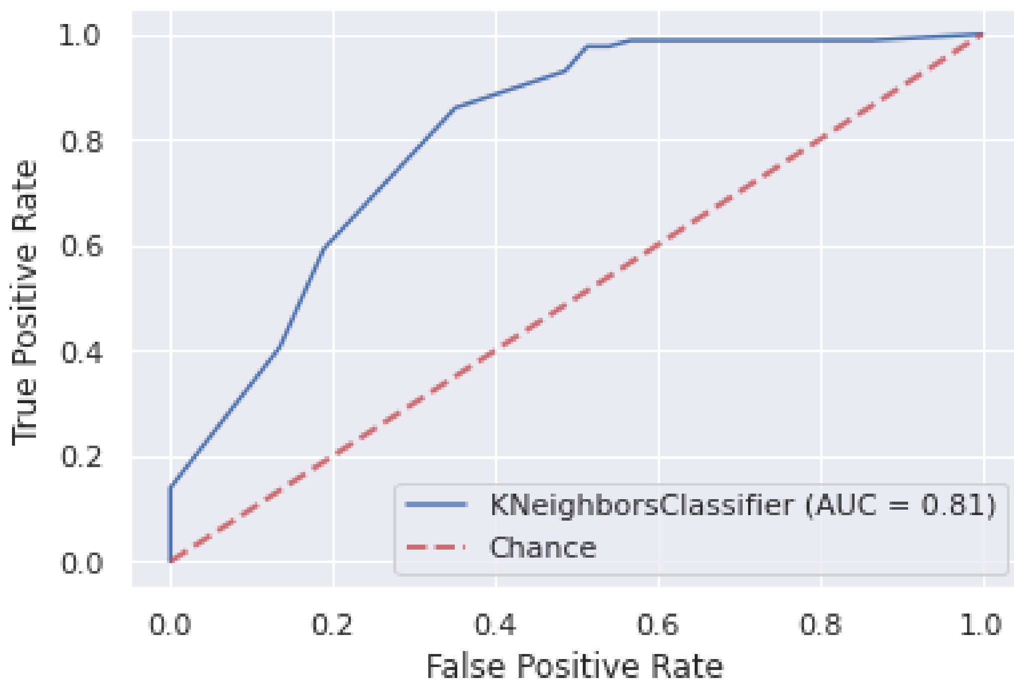
[29]:
```
from sklearn.neighbors import KNeighborsClassifier

knn_clf = KNeighborsClassifier()
knn_values = {"n_neighbors": list(range(2, 16, 2)),
              "algorithm": ["auto", "ball_tree", "kd_tree", "brute"],
              "leaf_size": [10, 20, 30, 40, 50],
              "p": [1, 2],
              "weights": ["uniform", "distance"]}

knn_clf = get_gscv(knn_clf, knn_values)
```

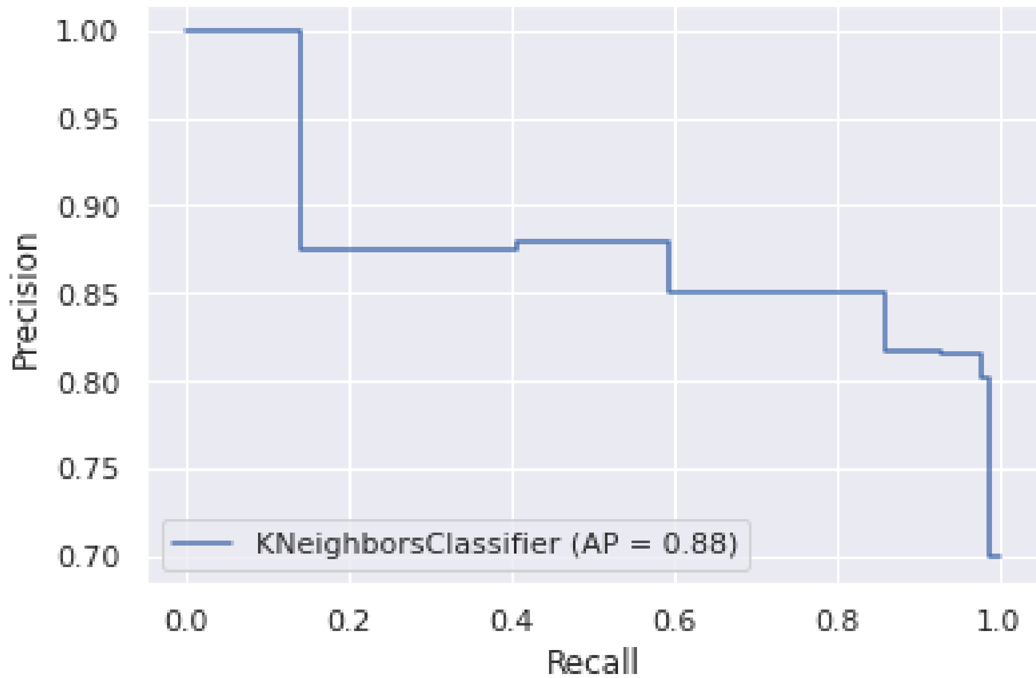
```
Fitting 3 folds for each of 560 candidates, totalling 1680 fits
[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=-2)]: Done 250 tasks      | elapsed:    2.1s
[Parallel(n_jobs=-2)]: Done 1450 tasks      | elapsed:   11.5s
```

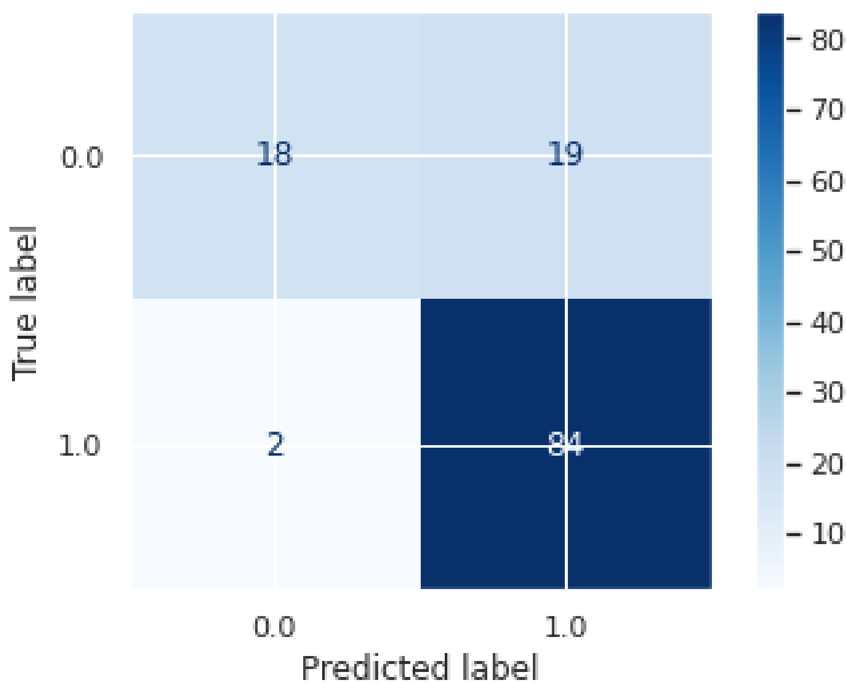
```
Best parameters: {'algorithm': 'auto', 'leaf_size': 10, 'n_neighbors': 12, 'p':
2, 'weights': 'uniform'}, with F1 score of 0.83
[Parallel(n_jobs=-2)]: Done 1680 out of 1680 | elapsed:   12.8s finished
```

[30]:
```
evaluate(knn_clf, test_X, test_y)
roc(knn_clf, test_X, test_y)
prc(knn_clf, test_X, test_y)
confusion(knn_clf, test_X, test_y)
plt.show()
```

```
RMSE: 0.4132
Accuracy: 0.803 ± 0.189
F1 Score: 0.81
```




1.00
0.95
0.90
0.85
0.80
0.75
0.70
Precision
0.0
0.2
0.4
0.6
0.8
1.0
Recall
KNeighborsClassifier (AP = 0.88)


0.0
1.0
True label
18
19
2
84
0.0
1.0
Predicted label
80
70
60
50
40
30
20
10

0.6 6. Support Vector Machine

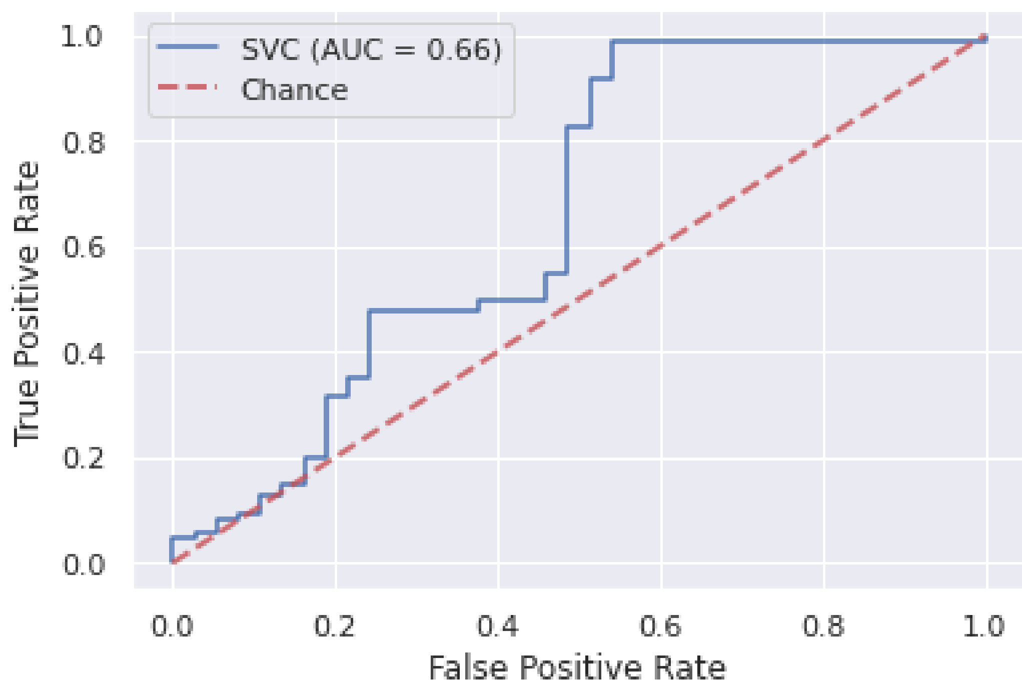
Support Vector Machines, abbr. SVM, is a supervised-learning algorithm used mainly for binary classification, although it is possible to use for multi-class classification by combing several SVMs. It creates hyperplanes in a multi-dimensional feature space, which are then used for generalization and classifications of data points. The best performing hyperplanes are those having the biggest maximum margin, i.e. the closest data points from both classes are as far as possible. In order to transform input data into a desired form, SVM uses so called kernel functions, which return the inner product between two points in a suitable feature space.

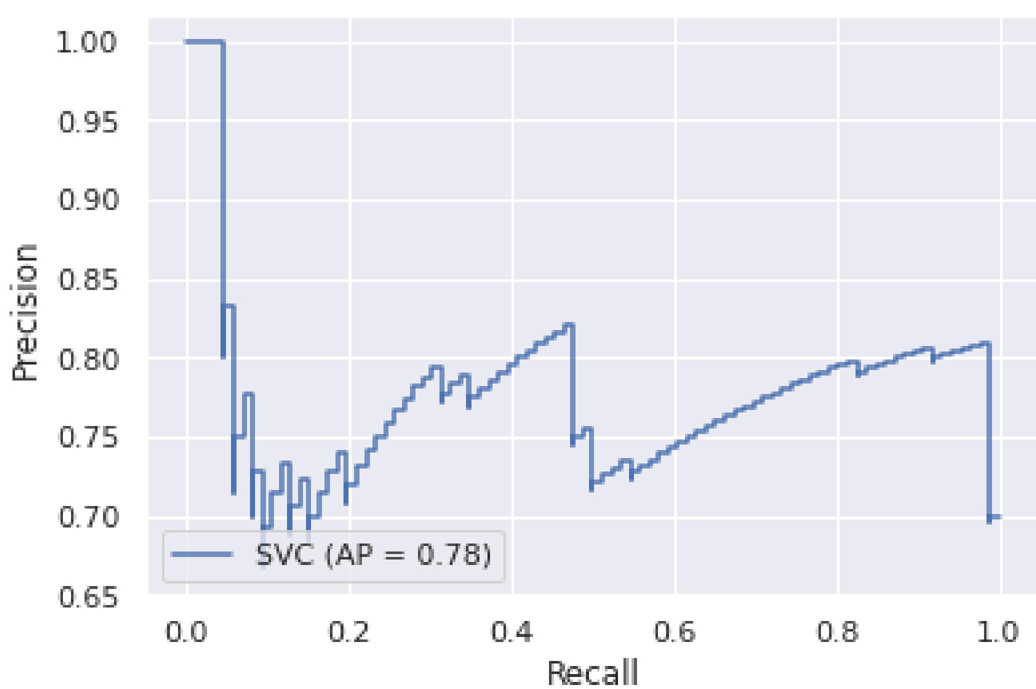
[31]:
```
from sklearn.svm import SVC

svc_clf = SVC()
svc_values = {"C": [.8, 1.0, 1.2, 1.5, 2.0],
              "kernel": ["linear","poly","rbf","sigmoid"],
              "degree": [2, 3, 4, 5],
              "gamma": ["scale", "auto"],
              "coef0": [.0, .5, 1.0, 1.5]}

svm_clf = get_gscv(svc_clf, svc_values)
```

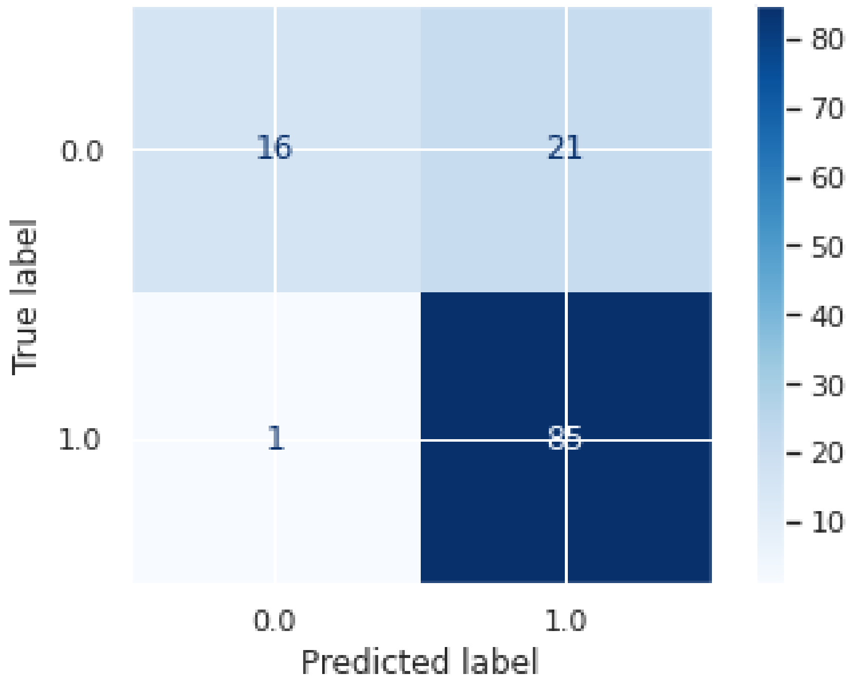
```
Fitting 3 folds for each of 640 candidates, totalling 1920 fits
[Parallel(n_jobs=-2)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=-2)]: Done 250 tasks      | elapsed:    1.8s
[Parallel(n_jobs=-2)]: Done 1450 tasks      | elapsed:   11.0s
Best parameters: {'C': 0.8, 'coef0': 0.0, 'degree': 2, 'gamma': 'scale',
'kernel': 'linear'}, with F1 score of 0.82
[Parallel(n_jobs=-2)]: Done 1920 out of 1920 | elapsed:   14.6s finished
```

[33]:
```
evaluate(svm_clf, test_X, test_y)
roc(svm_clf, test_X, test_y)
prc(svm_clf, test_X, test_y)
confusion(svm_clf, test_X, test_y)
plt.show()
```

```
RMSE: 0.4229
Accuracy: 0.819 ± 0.198
F1 Score: 0.80
```


SVC (AUC = 0.66)
Chance
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


SVC (AP = 0.78)
Precision
Recall
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.0
0.2
0.4
0.6
0.8
1.0



0.7 7. Deep Neural Network

Deep neural networks (DNNs) are artifical neural (ANNs) networks with several hidden layers. Each layer is a fixed number of artificial neurons, which accept an input, process it, and send it to the next layer. The layers are organized followingly:

input layer → hidden layers → output layer.

Each layer has an activation function, whose choice greatly influences the overall performance. In classification tasks, the output layer yields the final class labels. We will use `Sequential model` from `Keras` as our DNN.

For the activation functions we will stick with ReLU or Recrified Linear Units. These are nearly linear functions commonly used in DNNs and provide the best results. Leaky ReLU may be used as well.

We will be choosing either Adam, Stochastic Gradient Descent or RMSProp optimizer.

Batch size will remain constant 32 and epochs 10-15, since these numbers offer the best results for the time spent learning.

We have experimented with dropout a bit, but found little to no difference when using it, so it will be kept at 0

[34]:
```
import tensorflow as tf
import itertools
import itertools
```

```python
import gc

import keras.backend as K
from keras.optimizers import Adam, SGD, RMSprop
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from keras.layers import Dense, Dropout
from sklearn.metrics import classification_report, confusion_matrix
from keras.wrappers.scikit_learn import KerasClassifier
```

Using TensorFlow backend.

[35]:
```python
layer_sizes = [[[8, 16, 32, 64] for _ in range(size)] for size in range(6, 7)]
layer_combinations = list(itertools.chain.from_iterable(map(lambda sublist:␣
↪list(itertools.product(*sublist)), layer_sizes)))
del layer_sizes

gc.collect()
gc.enable()
```

[36]:
```python
def build_net(optim, layers, lr, dropout, **kwargs):
    K.clear_session()
    model = Sequential()
    model.add(Dense(layers[0], input_shape=(train_X.shape[1],),␣
↪activation='relu'))
    model.add(Dropout(dropout))
    for layer in layers[1:]:
        model.add(Dense(layer, activation='relu'))
        model.add(Dropout(dropout))
    model.add(Dense(2, activation='softmax'))
    model.compile(loss='categorical_crossentropy',␣
↪optimizer=optim(learning_rate=lr), metrics=['accuracy'])
    return model
```

```python
net_clf = KerasClassifier(build_fn=build_net, verbose=0)

layer_sizes = [[[32, 64, 128] for _ in range(size)] for size in range(3, 5)]
layer_combinations = list(itertools.chain.from_iterable(map(lambda sublist: list(itertools.proc

net_values = {"optim": [Adam, SGD, RMSprop], "epochs": [10], "batch_size": [32], "layers": laye

es = EarlyStopping(monitor='loss', min_delta=0, patience=2, verbose=0, mode='auto')

dnn_clf = get_gscv(net_clf, net_values, callbacks=[es])
```

Fitting 3 folds for each of 2880 candidates, totalling 8640 fits

1. Best parameters: {'batch_size': 32, 'dropout': 0.1, 'epochs': 10, 'layers': (32, 64, 32, 64), 'lr': 0.004, 'optim': <class 'keras.optimizers.RMSprop'>}, with F1 score of 0.78

2. Best parameters: {'batch_size': 32, 'dropout': 0.0, 'epochs': 15, 'layers': (8, 32, 8, 32), 'lr': 0.001, 'optim': <class 'keras.optimizers.RMSprop'>}, with F1 score of 0.77

3. Best parameters: {'batch_size': 32, 'dropout': 0.1, 'epochs': 12, 'layers': (64, 64, 64, 64, 32, 16), 'lr': 0.0003, 'optim': <class 'keras.optimizers.Adam'>}, with F1 score of 0.85

[ ]:
```
ohe = OneHotEncoder()

nn_train_y = ohe.fit_transform(y_train.to_numpy().reshape(-1, 1))
nn_test_y = ohe.transform(y_test.to_numpy().reshape(-1, 1))
```

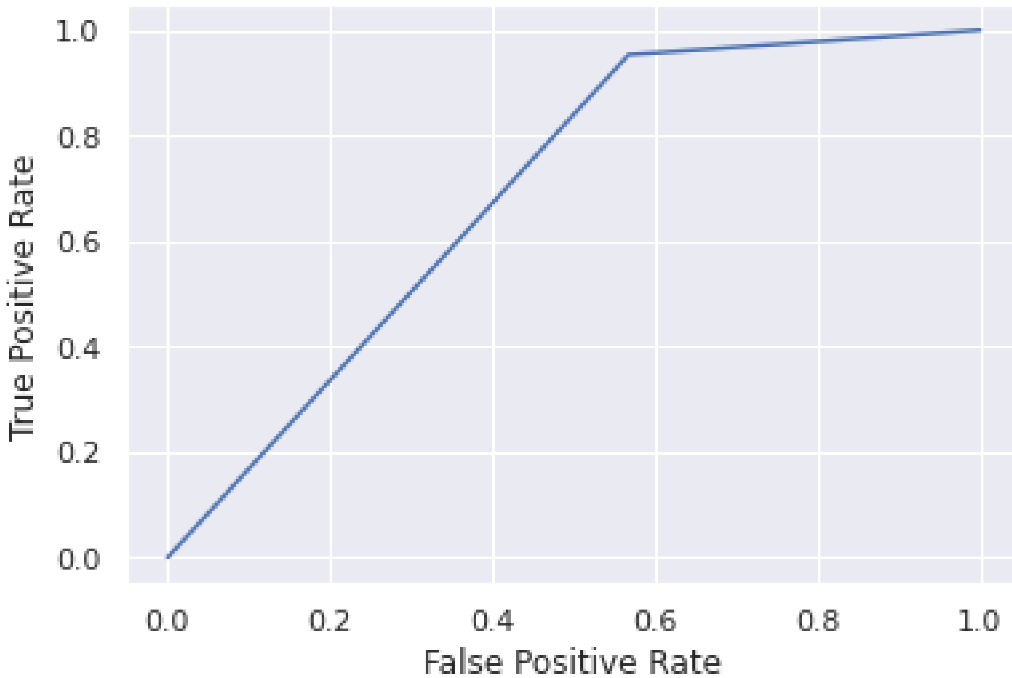
[38]:
```
# best_args = {'batch_size': 32, 'dropout': 0.1, 'epochs': 15, 'layers': (32,␣
↪32, 8), 'lr': 0.0004, 'optim': RMSprop}
# best_args = {'batch_size': 32, 'dropout': 0.1, 'epochs': 12, 'layers': (64,␣
↪64, 64, 64, 32, 16), 'lr': 0.0003, 'optim': Adam}
# best_args = {'batch_size': 32, 'dropout': 0.0, 'epochs': 15, 'layers': (32,␣
↪32, 16, 8), 'lr': 0.0003, 'optim': Adam}
# best_args = {'batch_size': 32, 'dropout': 0.0, 'epochs': 12, 'layers': (64,␣
↪64, 64, 16, 32, 64), 'lr': 0.0003, 'optim': Adam}
best_args = {'batch_size': 32, 'dropout': 0.0, 'epochs': 20, 'layers': (256,␣
↪128, 64, 64, 32), 'lr': 0.0003, 'optim': Adam}

dnn_clf = KerasClassifier(build_fn=build_net, verbose=0, **best_args)

data = dnn_clf.fit(train_X, train_y, epochs=15, batch_size=32)
dnn_clf.model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               5376
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
```

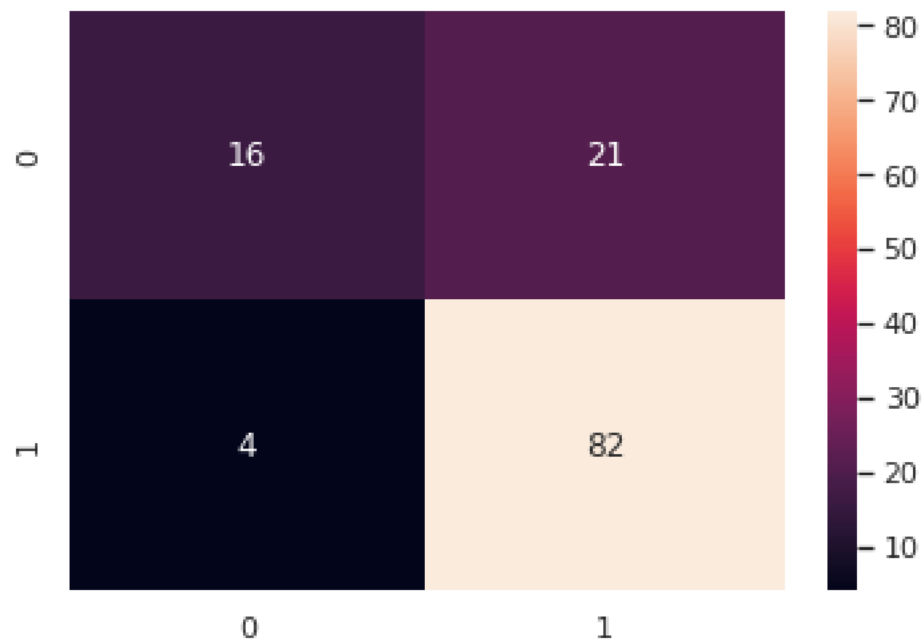
```
dense_4 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout_5 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 66
=================================================================
Total params: 52,834
Trainable params: 52,834
Non-trainable params: 0
_________________________________________________________________
```

[55]:

```
from sklearn.metrics import roc_curve

true_y_labels = np.argmax(nn_test_y, axis=1)

predicted_y = dnn_clf.predict(test_X)

roc_curve(true_y_labels, predicted_y)
fpr, tpr, thresholds = roc_curve(true_y_labels, predicted_y)

roc_data = pd.DataFrame({"False Positive Rate": fpr, "True Positive Rate": tpr})

sns.lineplot(x="False Positive Rate", y="True Positive Rate", data=roc_data)
plt.show()

history_df = pd.DataFrame(data=data.history, columns=data.history.keys())
sns.lineplot(legend='full', y=history_df['loss'], x=range(len(data.
 ↪history['loss'])), label='loss')
sns.lineplot(legend='full', y=history_df['accuracy'], x=range(len(data.
 ↪history['accuracy'])), label='accuracy')
plt.show()

evaluate(dnn_clf, test_X, test_y)
sns.heatmap(confusion_matrix(true_y_labels, predicted_y), annot=True)
plt.show()

print('\nClassification Report')
target_names = ["Y", "N"]
print(classification_report(true_y_labels, predicted_y,␣
 ↪target_names=target_names))
```




```
RMSE: 0.4508
Accuracy: 0.812 ± 0.214
F1 Score: 0.78
```



0.8 8. Evaluation

It is easy to see, that we have moved far beyond the perfomance of the baseline model. Therefore, we could assume our project reached its goal.

From the evaluation metrics it seems, that *deep neural network*, *KNN*, and *SVM* performed nearly equally. Their accuracy exceeded 80%. This is quite surprising as *KNN* can be considered as the most simple from all 4 models and yet it kept pace with them. On the other side of the spectrum is *decision tree classifier*, which had the worst evaluation metrics from all 4 models.

And finally, the winner's podium:

1. Deep Neural Network, KNN Classifier, Support Vector Machine
2. Decision Tree Classifier
3. Dummy classifier

Though keep in mind that with such small dataset the performance of all models may be influenced by the random state quite a bit.

The performance of all models could be improved in certain models (such as the DNN classifier) by using weighted samples as the dataset is quite imbalanced.

The accuracy has quite a large differences between positive and negative samples. This is most likely caused by the fact that sampling for model training is not stratified whereas accuracy is computed using stratified cross validation.

All models seemed to have problems with recognising true negatives. In all cases the number of false negatives was greater than the number of true negatives. Though this could be due to imbalanced dataset as well.

0.9 9. Conclusion

We have explored and preprocessed the dataset. From the computational side, training of the models and tuning of their hyperparameters did not take too long, in average about 35sec per model, with neural network being an exception, as it was trained with several epochs for each parameter search. Even though the dataset did not offer many records, we can conclude that the models performed overall quite well.