Introduction to Regression with statsmodels in Python
by datacamp

Regression models are a class of statistical models that let you explore the relationship between a response variable and some explanatory varialbes
response variable is also called the dependent variable or the ‘y’ variable
- the variable that you want to predict
explanatory variable(s) also called the independent variable or the ‘x’ variable
- the variables that explain how the response variable will change

linear regression - response variable is numeric
**the ‘y’ variable is numeric
logistic regression - repsonse variable is logical
**ie it takes true or false values

simple linear/logistic regression only takes in one explanatory (x) variable

reminder regplot() in seaborn adds a trend line calculated using linear regression

Python packages for regression
statsmodels - optimized for insight (focus in this course)
scikit-learn - optimized for prediction

Defining feature of trend lines in linear regression is that they are ‘straight’
straight lines are defined by two things
Intercept - the y value at the point when x is zero
Slope - the amount the y value increases if you increase x by one
Equation putting it together
y = intercept + slope * x

Estimating the slope
first estimate the intercept
- where does the trend line intersect the y-axis

second estimate the slope
- need two points to do this
- choose points near grid lines to make visualizing easier
- then calculate the change in y values between the points
- then calculate the change in x values between the points
- then divide y by x
- this is our estimated slope

example
from statsmodel.formula.api import ols
#'ols' stands for ordinary least squares - this is a type of regression
mdl_payment_vs_claims = ols('total_payment_sek ~ n_claims',
data=swedish_motor_insurance)
#the ols function takes in two arguments:
- the first is a formula > response variable (y) is written to the left of the tilde, and the explanatory variable (x) is written to the right
- the second is the data argument and takes in the dataset
mdl_payment_vs_claims = mdl_payment_vs_claims.fit()
#fit method calls on an instantiated model object and estimates the model parameters based on the provided data
#it fits the model to the data by finding the parameter values that best align with the observed data
#it does this according to the model's estimation technique - in this example off least squares
#once fit() completes you can access specific attributes depending on the model type
#such as estimated coefficients, statistical measures of fit, residuals, hypothesis tests, etc
print(mdl_payment_vs_claims.params)
#the params attribute contains the model's parameters
#it will result in two coefficients - y-intercept and the slope of the straight line

Categorical explanatory variables
scatterplots for numerical data
histograms for categorical data
use the sns.displot(data,x,col,col_wrap,bins)
x is the variable of interest
col defines the variable that you want to split on

Summary statistics
example
summary_stats = df.groupby(variable to split on)[variable of interest].mean()
#variable to split on > ie how you intend to subset your data
summary_stats = fish.groupby('species')['mass_g').mean()
print(summary_stats)

Run a linear regression
using mass as response variable
using species as explanatory variable
from statsmodels.formula.api import ols
mdl_mass_v_species = ols('mass_g ~ species', data=fish).fit()

print(mdl_mass_v_species.params)
**output #for this example there are 4 fish
> intercept of one of the fish, then three additional coeffs for the other three species
#these values are relative to the intercept and this is why you can see negative numbers
#in this example appears breem grams avg (intercept) is 617g > perch coeff is -235 which appears to say perch gram avg is 617 - 235
**this is confusing, particularly if just dealing with simple linear regression
**fix it this way > ols('mass_g ~ species + 0', data=fish).fit()
#this now makes all coeffs relative to zero
**also means that we are fitting a linear regression without an intercept term
**now our output just shows all four species with their mean mass in grams

Making predictions
**the big benefit of running models rather than simply calculating descriptive statistics is that models let you make predictions
The general concept - if I set x to these values, what value would y have?

Data on explanatory values to predict
to start choose some values for the explanatory variables
to create new explanatory data, we need to store our explanatory variables of choice in a pandas DataFrame
use a dictionary to specify the columns
this example we are using one explanatory variable - length of the breem fish
explanatory_data = pd.DataFrame({'length_cm': np.arange(20,41)})
#use np.arange to specify an interval of values (takes in the start and end of the range or interval as arguments)
#here we are specifiying a range of 20-40cm

Next call predict()
print(mdl_mass_vs_length.predict(explanatory_data))
#predict() function returns a Series of predictions, one for each row of the explanatory data
**predictions as a single column are not that helpful to work with
easier to work with in a DF alongside the explanatory variables
prediction_data =
explanatory_data.assign(mass_g=mdl_mass_vs_length.predict(explanatory_data))
print(prediction_data)
#the assign() method returns a new object with all original columns in addition to new ones
#inside assign() you add a new column named after the response variable (in our example it is mass_g)

**the resulting DF contains both the explanatory variable and the predicted response

Showing predictions
to plot multiple layers we set a matplotlib figure object called fig before calling regplot and scatterplot
**this will allow us to plot both graphs on the same figure when we call plt.show()
***notice that the predictions lie exactly on the trend line

Extrapolating
making predictions outside the range of observed data
***be aware this can sometimes be appropriate but can lead to misleading or ridiculous results
**the key here is to understand the context of your data

Nice reference:
explanatory_data = pd.DataFrame({"explanatory_var": list_of_values})
predictions = model.predict(explanatory_data)
prediction_data = explanatory_data.assign(response_var=predictions)

example
```
# Create a new figure, fig
fig = plt.figure()

sns.regplot(x="n_convenience",
            y="price_twd_msq",
            data=taiwan_real_estate,
            ci=None)
# Add a scatter plot layer to the regplot
sns.scatterplot(data=prediction_data,
                x='n_convenience',
                y='price_twd_msq',
                color='r')

# Show the layered plot
plt.show()
```

Working with model objects
.fittedvalues attribute
predictions on the original dataset
example
```
print(mdl_mass_vs_length.fittedvalues)
```
**fittedvalues attribute is essentially a shortcut for taking the explanatory variable

columns from the dataset, then feeding them to the predict function
**equivalent to doing this:
explanatory_data = bream['length_cm']
print(mdl_mass_vs_length.predict(explanatory_data))

Residuals
are a measure of inaccuracy in the model fit
**there is one residual for each row of the dataset
actual response values minus predicted response values
in this example it is the mass of the breem minus the fitted values
print(mdl_mass_vs_length.resid)
equivalent to this:
print(bream['mass_g'] - mdl_mass_vs_length.fittedvalues)

Explaining the summary() method

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 mass_g   R-squared:                       0.878
Model:                            OLS   Adj. R-squared:                  0.874
Method:                 Least Squares   F-statistic:                     237.6
Date:                Thu, 29 Oct 2020   Prob (F-statistic):           1.22e-16
Time:                        13:23:21   Log-Likelihood:                -199.35
No. Observations:                  35   AIC:                             402.7
Df Residuals:                      33   BIC:                             405.8
Df Model:                           1
Covariance Type:            nonrobust
```

```
                 coef    std err          t      P>|t|      [0.025      0.975]
<------------------------------------------------------------------------------
Intercept  -1035.3476    107.973     -9.589      0.000   -1255.020    -815.676
length_cm     54.5500      3.539     15.415      0.000      47.350      61.750
==============================================================================
Omnibus:                        7.314   Durbin-Watson:                   1.478
Prob(Omnibus):                  0.026   Jarque-Bera (JB):               10.857
Skew:                          -0.252   Prob(JB):                      0.00439
Kurtosis:                       5.682   Cond. No.                         263.
```

example
Get the coefficients of mdl_price_vs_conv

```
coeffs = mdl_price_vs_conv.params

# Get the intercept
intercept = coeffs[0]

# Get the slope
slope = coeffs[1]

# Manually calculate the predictions
price_twd_msq = intercept + (slope * explanatory_data)
print(price_twd_msq)

# Compare to the results from .predict()
print(price_twd_msq.assign(predictions_auto=mdl_price_vs_conv.predict(explanato
ry_data)))
```

Regression to the mean
a property of the data, not a type of model
linear regression can be used to quantify its effect
concept

1. response value = fitted value + residual
2. ie “the stuff you explained” + “the stuff you couldn’t explain”
3. residuals exist due to problems in the model and fundamental randomness
4. extreme cases are often duet to randomness
5. regression to the mean means extreme cases don’t persist over time

the quintessential example is that tall fathers on average tend to have sons with heights that regress to the average male’s height more than continued or increased extreme tallness - the same could be said for short fathers

```
# Create a new figure, fig
fig = plt.figure()

# Plot the first layer: y = x
plt.axline(xy1=(0,0), slope=1, linewidth=2, color="green")

# Add scatter plot with linear regression trend line
sns.regplot(x='return_2018', y='return_2019', data=sp500_yearly_returns, ci=None)

# Set the axes so that the distances along the x and y axes look the same
plt.axis('equal')

# Show the plot
plt.show()
```

Transforming variables
when to transform?
when there is no straight-line relationship between the response variable and the explanatory variable
can transform response variables or expanatory variables or both
types of transforming - squaring, cubing, square root
the transformation depends on the data
example
bream grows in length while the perch grows in length, width, and height
original data shows a nice linerar relationship with mass and length with the bream but not the perch
here we could re-evaluate the perch relationship after cubing the length
instead of 3-dimensional growth, we are placing that growth into length
create a new variable to do this and replace the old length variable
this new transformation and fitting shows non-linear predictions on a linear model (points curve with data now)

another example
when a model distribution is skewed right (meaning a majority of the data falls on the bottom-left of the graph)
this can make it difficult to assess the fit
by transforming both the variable with square roots, the data becomes more spread out throughout the plot
**here a key step post running the model is to undo the square root by squaring the predicted responses
this is called back transformation

Example
```
# Create sqrt_dist_to_mrt_m
taiwan_real_estate["sqrt_dist_to_mrt_m"] =
np.sqrt(taiwan_real_estate["dist_to_mrt_m"])

# Run a linear regression of price_twd_msq vs. square root of dist_to_mrt_m using taiwan_real_estate
mdl_price_vs_dist = sns.regplot(x='sqrt_dist_to_mrt_m', y='price_twd_msq', data=taiwan_real_estate)

# Print the parameters
sqrt_dist_to_mrt_m = ols('price_twd_msq ~ sqrt_dist_to_mrt_m', data=taiwan_real_estate).fit()
print(sqrt_dist_to_mrt_m.params)
```

Example

```
ad_conversion["qdrt_n_impressions"] = ad_conversion["n_impressions"] ** 0.25
ad_conversion["qdrt_n_clicks"] = ad_conversion["n_clicks"] ** 0.25

mdl_click_vs_impression = ols("qdrt_n_clicks ~ qdrt_n_impressions",
data=ad_conversion, ci=None).fit()

explanatory_data = pd.DataFrame({"qdrt_n_impressions": np.arange(0, 3e6+1,
5e5) ** .25,
                                 "n_impressions": np.arange(0, 3e6+1, 5e5)})

# Complete prediction_data
prediction_data = explanatory_data.assign(
    qdrt_n_clicks = mdl_click_vs_impression.predict(explanatory_data),
qdrt_n_impressions = mdl_click_vs_impression.predict(explanatory_data) ** 4)

# Print the result
print(prediction_data)

# Back transform qdrt_n_clicks
prediction_data["n_clicks"] = prediction_data["qdrt_n_clicks"] ** 4

# Plot the transformed variables
fig = plt.figure()
sns.regplot(x="qdrt_n_impressions", y="qdrt_n_clicks", data=ad_conversion,
ci=None)

# Add a layer of your prediction points
sns.scatterplot(data=prediction_data, x='qdrt_n_impressions', y='qdrt_n_clicks',
color='r')
plt.show()
```

Quantifying model fit
how strong is the linear relationship?
first metric
coffeicient of determination - also called r-squared or R-squared
why sometime lowercase r and other times for uppercase R
old school reasons
r for simple linear regression
R for when you have more than one explanatory variable
**coefficient of determintation is the proportion of the variance in the response variable that is predictable from the eplanatory variable
score of 1 means a perfect fit

score of 0 means the worst possible fit or your model is no better than randomness
**dataset plays a big part in what constitutes a “good” score
0.5 for things that are hard to predict may be considered a good score
where a 0.9 in something that is easy to predicat may be considered a poor fit
R-squared can be found in the summary() method
by itself
print(mdl_example.rsquared)
***for simple linear regression, the interpretation of the coefficient of determination is straightforward; it is simply the correlation between the explanatory and response variables, squared.
coeff_determination = bream[‘length_cm’].corr(bream[‘mass_g’]) ** 2
print(coeff_determination)
**will give you same answer as print(mdl_bream.rsquared)

2nd metric
Residual standard error (RSE)
residual is the difference between a predicted value and an observed value
RSE roughly speaking is a measure of the typical size of the residuals
ie how much the predictions are typically wrong
a typical difference between a prediction and an observed response
**has the same unit as the response (y) variable
not contained in the summary() method
Less commonly used, but related metric is the mean squared error (MSE)
MSE is the squared residual standard error
RSE can indirectly be retrieved from the mse_resid attribute
this contains the mean squared error of the residuals
mse = mdl_bream.mse_resid
print(‘mse: ‘, mse)
then
rse = np.sqrt(mse)
print(‘rse: ‘, rse)

Calculating RSE yourself
residuals_sq = mdl_bream.resid ** 2
resid_sum_of_sq = sum(residuals_sq)
deg_freeedom = len(bream.index) - 2
print(‘deg freedom: ‘, deg_freedom)
**degrees of freedom equals the number of observations minus hte number of model coefficients
rse = np.sqrt(resid_sum_of_sq/deg_freedom)
print(‘rse :’, rse)
**in this example output is 74g, which essentially states the difference between

predicted bream masses and observed bream masses is typically about 74g

**number of model coefficients depends on the complexity of the model
in simple linear regression you have two coefficients the intercept (also known as the bias term) and the slope of the single independent variable
for multiple linear regression (ie multiple independent variables), there is one coefficient for each independent variable, plus the intercept

another related metric is the root-mean-square error (RMSE)
**calculated in the same way, except you don't subtract the number of coefficients in the second to last step
*performs same task as residual standard error, namely quantifying how inaccurate the model predictions are
But is worse for comparisons between models
**you should use RSE instead
RMSE example

```
residuals_sq = mdl_bream.resid ** 2
resid_sum_of_sq = sum(residuals_sq)
n_obs = len(bream.index)
rmse = np.sqrt(resid_sum_of_sq/n_obs)
print('rmse :', rmse)
```

Visualizing model fit
residual properties of a good fit
-residuals are normally distributed
-the mean of the residuals is zero
plot fitted (x) vs residuals (y) - gives a good idea if a good fit
the LOWESS trend line should follow the y equals zero line on the plot
if it does then your residuals are meeting the above parameters

Q-Q plot
shows whether or not the residuals follow a normal distribution
x-axis shows quantiles from what would be the normal distribution
the y-axis shows the sample quantiles
if the points track along the straight line they are normally distributed

Scale-location plot
shows square root of the standardized residuals versus the fitted values
this plot shows whether the size of the residuals gets bigger or smaller
fitted values (x) vs sqrt(standardized residuals) (y)
ideally looking for a line that goes horizontally straight across
poor fit will have a not so straight line with significant ups and downs

Residuals v Fitted plot
sns.residplot(x='length_cm', y='mass_g', data=bream, lowess=True)
plt.xlabel('Fitted values')
plt.ylabel('Residuals')

Q-Q plot
from statsmodels.api import qqplot
qqplot(data=mdl_bream.resid, fit=True, line='45')
fit argument to True will compar the data quantiles to a normal distribution
line argument to 45, sets a 45 degree line (optional feature), may make it more readable

Scale-location plot
model_norm_residuals_bream =
mdl_bream.get_influence().resid_studentized_internal
model_norm_residuals_abs_sqrt_bream =
np.sqrt(np.abs(model_norm_residuals_bream))
#get_influence() method helps you extract the normalized residuals from the model
#np.abs() method takes the absolute values (remember residuals can be positive or negative)
now you can plot
sns.regplot(x=mdl_bream.fittedvalues, y=model_norm_residuals_abs_sqrt_bream, ci=None, lowess=True)
plt.xlabel('Fitted values')
plt.ylabel('Sqrt of abs val of stdized residuals')

Leverage
a measure of how extreme the explanatory variable values are

Influence - a type of 'leave one out' metric
measures how much the model would change if you left the observation out of the dataset when modeling

The mathematics can become challenging once there is more than one explanatory (independent) variable
to get the leverage and influence metrics
you have to retrieve the summary_frame() method
example
first you call the get_influence() method on the fitted model then call the summary_frame method
mdl_roach = ols('mass_g ~ length_cm', data=roach).fit()
summary_roach = mdl_roach.get_influence().summary_frame()

historically leverage was described in the so-called hat matrix
therefore you find the values of leverage are stored in the hat_diag column
roach[‘leverage’] = summary_roach[‘hat_diag’]
print(roach.head())
output > returns an array with as many values as there are observations
**in this example these leverage values indicates how extreme your roach lengths are

Cook’s distance
is the most common measure of influence
Recall that influence is based on the size of the residuals and the leverage
Cook’s distance is stored in the summary_frame() method as ‘cooks_d’
roach[‘cooks_dist’] = summary_roach[‘cooks_d’]
print(roach.head())
can find the most influential roaches
print(roach.sort_values(‘cooks_dist’, ascending=False))

example
# Create summary_info
summary_info = mdl_price_vs_dist.get_influence().summary_frame()

Add the hat_diag column to taiwan_real_estate, name it leverage
taiwan_real_estate["leverage"] = summary_info["hat_diag"]

Add the cooks_d column to taiwan_real_estate, name it cooks_dist
taiwan_real_estate['cooks_dist'] = summary_info['cooks_d']

# Sort taiwan_real_estate by cooks_dist in descending order and print the head.
print(taiwan_real_estate.sort_values('cooks_dist', ascending=False).head())

Logistic regression
example with binary (0 or 1) response variable
logical > True/False or 0/1
**ie y is either equal to 0 or equal to 1
*if you were to use linear regression you would get fractions as predictions, which could be looked at as probabilities
however, in these predictions you would negative probabilities or probabilities >1 which are both impossible
this is where logistic regression comes in
logistic for logistical
logistic regression models are a type of generalized linear model used when the response variable is logical
**where linear models result in predictions that follow a straight line, logistic

models result in predictions that follow a logistic curve (S-shape)
how to run
from statsmodels.formula.api import logit
mdl_churn_vs_recency_logit = logit('has_churned ~ time_since_last_purchase', data=churn).fit()
**same argument and fit format as 'ols' function
print(mdl_churn_vs_recency_logit.params)

Visualizing
#regplot will draw a logistic regression trend line as long as you set the logistic argument to True
example
sns.regplot(x='time_since_last_purchase', y='has_churned', data=churn, ci=None, logistic=True)
zooming out shows that the logistic regression curve never goes below zero or above one

Predictions and odds ratio
mdl_recency = logit('has_chruned ~ time_since_last_purchase', data=churn).fit()
explanatory_data = pd.DataFrame({'time_since_last_purchase': np.arange(-1, 6.25, 0.25)})
prediction_data = explanatory_data.assign(has_churned = mdl_recency.predict(explanatory_data))
#adding point predictions
sns.regplot(x='time_since_last_purchase', y='has_churned', data=churn, ci=None, logistic=True)
sns.scatterplot(x='time_since_last_purchase', y='has_churned', data=prediction_data, color='r')
plt.show()

Getting the most likely outcome
in place of calculating probabilities of a response, you can calculate the most likely response
ie if the prob is >0.5 they won't churn, <=0.5 they will churn
prediction_data = explanatory_data.assign(has_churned = mdl_recency.predict(explanatory_data))
prediction_data['most_likely_outcome'] = np.round(predicition_data['has_churned'])
plot as above but with y='most_likely_outcome' in the scatterplot

Odds ratios
the probability of something happening divided by the probability that it doesn't
odds_ratio = probability/(1-probability)

example
odds_ratio = 0.25(1-0.25) = 1/3
basically says the probability of an event happening is 0.25 so the event of it not happening is 0.75

Calculating the odds ratio
prediction_data['odds_ratio'] = prediction_data['has_churned'] / (1 - prediction_data['has_churned'])
divide the predicted response probability by one minus that number

Visualizing odds ratio
sns.lineplot(x='time_since_last_purchase', y='odds_ratio', data=prediciton_data)
plt.axhline(y=1, linestyle='dotted')
plt.yscale('log')
plt.show()
creating a plot with a continuous line
plt.axhline creates the odds ratio line, indicates where churning is just as likely as not churning
**in this example below the odds ratio line the chance of churning is less than the chance of not churning, above is opposite
a nice property of logistic regression odds ratio is that on on a log-scale, they change linearly with the explanatory variable
**this nice property of the logarithm of odds ratios means log-odds ratio is another common way of describing logitstic regression predictions
**this is also known as the "logit"

Calculating log odds ratio
prediction_data['log_odds_ratio'] = np.log(prediction_data['odds_ratio'])

Scale	Are values easy to interpret?	Are changes easy to interpret?	Is precise?
Probability	✔	✘	✔
Most likely outcome	✔✔	✔	✘
Odds ratio	✔	✘	✔
Log odds ratio	✘	✔	✔

Example - visualizing most likely outcome scale
Update prediction data by adding most_likely_outcome
prediction_data["most_likely_outcome"] =
np.round(prediction_data["has_churned"])

```
fig = plt.figure()

# Create a scatter plot with logistic trend line (from previous exercise)
sns.regplot(x="time_since_first_purchase",
            y="has_churned",
            data=churn,
            ci=None,
            logistic=True)

# Overlay with prediction_data, colored red
sns.scatterplot(x='time_since_first_purchase',
y='most_likely_outcome',data=prediction_data, color='red')

plt.show()
```

Example - showing how to visualize log odds ratio

```
# Update prediction data with log_odds_ratio
prediction_data["log_odds_ratio"] = np.log(prediction_data["odds_ratio"])

fig = plt.figure()

# Update the line plot: log_odds_ratio vs. time_since_first_purchase
sns.lineplot(x="time_since_first_purchase",
             y="log_odds_ratio",
             data=prediction_data)

# Add a dotted horizontal line at log_odds_ratio = 0
plt.axhline(y=0, linestyle="dotted")

plt.show()
```

Quantifying logistic regression fit
diagnostic plots ar less usefule in the logistic case
better to use confusion matrices for logistic regression
**a logical response variable leads to four possible outcomes

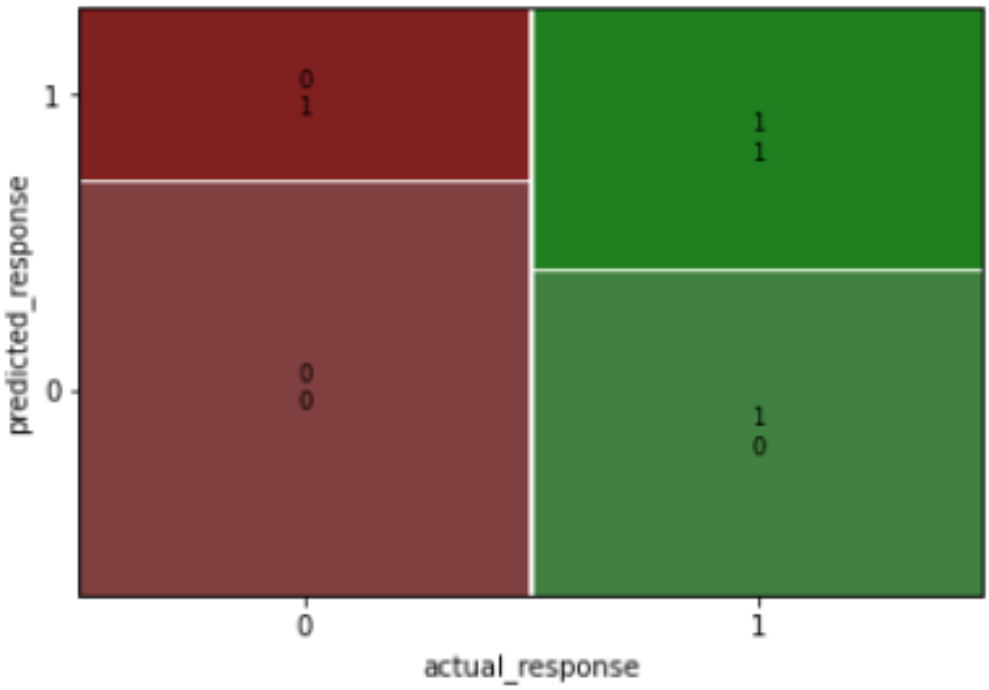
	predicted false	predicted true
actual false	correct	false positive
actual true	false negative	correct

Confusion matrix
getting counts of outcomes
actual_response = churn['has_churned']
predicted_response = np.round(mdl_recency.predict())
#these predicted values are probabilities
#to get the most likely outcome we need to round to 0 or 1
outcomes = pd.DataFrame({'actual_response': actual_response,
'predicted_response': predicted_response})
print(outcomes.value_counts(sort=False)
this was the long way

**short way
conf_matrix = mdl_recency.pred_table()
print(conf_matrix)
this returns an array

Visualizing the confusion matrix
from statsmodels.graphics.mosaicplot
import mosaic
mosaic(conf_matrix)
interpretation
width of each column is proportional to the fraction of observations in each category of actual values
then each column displays the fraction of predicted observations with each value

true negative	false positive
false negative	true positive



Accuracy
is the proportion of correct predictions
accuracy = (TN + TP) / (TN + FN + FP + TP)
how to code
TN = conf_matrix[0,0]
TP = conf_matrix[1,1]
FN = conf_matrix[1,0]
FP = conf_matrix[0,1]
acc = (TN + TP) / (TN + FN + FP + TP)
higher accuracy is better

Sensitivity
the proportion of true positives
sensitivity = TP / (FN + TP)
higher sensitivity is better

Specificity
the proportion of true negatives
specificity = TN / (TN + FP)
higher specificity is better

**often there is a trade-off where improving specificity will decrease sensitivity, vis-a-versa