Predicting Mortgage Delinquency Risk

Contents

- WARNING
- Gradescope Autograding
- Data Cleaning and Organization
- Modeling Delinquency Risk
- Now To The Future

You have been hired by a mortgage servicing firm (a company that buys mortgages and then collects mortgage payments from homeowners) to build a model to answer the question:

Given all available information about a newly issued mortgage, what is the likelihood that the mortgage will enter delinquency (the homeowner will be at least 30 days late on a mortgage payment) during the first two years of the mortgage?

The servicer's hope, obviously, is to differentiate between mortgages to try and purchase (those that will be consistently paid) and mortgages they wish to avoid.

For this task, you have been given REAL data on a sample of all US Standard single family home mortgages purchased or insured by Freddie Mac in a single calendar year along with payment data from that and two subsequent years.

WARNING

This is a longer assignment than many you are accustomed to if you were in IDS 720. Please start early!

Gradescope Autograding

Please follow all standard guidance for submitting this assignment to the Gradescope autograder, including storing your solutions in a dictionary called `results` and ensuring your notebook runs from the start to completion without any errors.

For this assignment, please name your file `exercise_passive_prediction.ipynb` before uploading.

You can check that you have answers for all questions in your `results` dictionary with this code:

```
assert set(results.keys()) == {
    "ex2_merge_type",
    "ex5_num_mortgages",
    "ex5_share_delinquent",
    "ex7_num_obs",
    "ex10_predicted_delinquent",
    "ex11_share_delinquent_weighted",
    "ex13_optimal_threshold",
    "ex14_normalized_value",
    "ex15_num_obs",
    "ex15_share_delinquent",
    "ex16_final_return_pct",
    "ex16_normalized_value_2007",
}
```

Submission Limits

Please remember that you are **only allowed THREE submissions to the autograder.** Your last submission (if you submit 3 or fewer times), or your third submission (if you submit more than 3 times) will determine your grade Submissions that error out will **not** count against this total.

Good Notebook Practices

Please also review and follow all Good Jupyter Notebook Practices guidelines. They ARE grade relevant.

Data Cleaning and Organization

Data for this exercise can be found here. This folder includes both the data to be used and documentation, though you can find supplemental documentation here.

The only modifications I've made to this data are:

- Subset to mortgages taken out for purchase of a property,
- With first payments due in the quarter of origination or the first quarter after origination (the vast majority of loans have first payments due the month after origination, so this just gets rid of some very weird mortgages).
- I have also excluded mortgages for which origination data is available but servicing data is not available in the two years following the year of origination.
- I also subset the data on servicing to the first 24 months after the mortgages first payment is due, so for each mortgage you will only have data on what happened in the first 24 months of its life.

However, post-subsetting I have done my best to convert the data back to the original format to make your experience working with the data as authentic as possible.

Exercise 1

Begin by loading both:

- the mortgage origination file (`sample_orig_2004_standard_mortgages.txt.zip`). This *should* contain information on all mortgages issued in 2004, along with non-time varying features of these mortgages (the initial amount, the credit score of the applicant, etc.), and
- the servicing data (`sample_svcg_2004_threeyears_standard_mortgages.txt.zip`). This contains monthly records of all recorded payments (or non-payments) for all mortgages issued in 2004 during the calendar years of 2004, 2005, and 2006. As noted above, this has also been subset to only include the first 24 months after the first payment was due (though Freddie Mac has some data cleanliness issues, so sometimes there will be less than 24 records covering that first 24 months).

So the autograder can see the data, be sure to load it directly from a URL (don't download and load from your own system).

Because this is **real** data, it has some issues (even beyond what I've cleaned above). While I generally love to leave students to work through this stuff, this is a long exercise, so here are a couple tips:

- The data is zip compressed. When you gives pandas a zip file that has only one thing in the zip archive, it will *usually* infer what's going on and decompress it without help. However, if the file name or URL you pass to pandas does not end in `.zip`, this automatic inference will fail and you will need to use the `compression` keyword to explicitly tell pandas the file is zip compressed.
- When you load the data, you will see it does not have column names. You will likely need to reference the documentation to figure out appropriate column names.
    - pandas will automatically treat the first row of a dataset as column names, not data. When working with a dataset that lacks column names, not only do you have to set the column names (so you know the meaning of each column), you also have to make sure pandas doesn't treat the first row as labels and not data (effectively dropping it).

```
import pandas as pd
import numpy as np

pd.set_option("mode.copy_on_write", True)
```

```
```

```
origination_colnames = [
    "Credit Score",
    "First Payment Date",
    "First Time Homebuyer Flag",
    "Maturity Date",
    "Metropolitan Statistical Area (MSA) Or Metropolitan Division",
    "Mortgage Insurance Percentage (MI %)",
    "Number of Units",
    "Occupancy Status",
    "Original Combined Loan-to-Value (CLTV)",
    "Original Debt-to-Income (DTI) Ratio",
    "Original UPB",
    "Original Loan-to-Value (LTV)",
    "Original Interest Rate",
    "Channel",
    "Prepayment Penalty Mortgage (PPM) Flag",
    "Amortization Type (Formerly Product Type)",
    "Property State",
    "Property Type",
    "Postal Code",
    "Loan Sequence Number",
    "Loan Purpose",
    "Original Loan Term",
    "Number of Borrowers",
    "Seller Name",
    "Servicer Name",
    "Super Conforming Flag",
    "Pre-HARP Loan Sequence Number",
    "Program Indicator",
    "HARP Indicator",
    "Property Valuation Method",
    "Interest Only (I/O) Indicator",
    "Mortgage Insurance Cancellation Indicator",
]
```

```
service_colnames = [
    "Loan Sequence Number",
    "Monthly Reporting Period",
    "Current Actual UPB",
    "Current Loan Delinquency Status",
    "Loan Age",
    "Remaining Months to Legal Maturity",
    "Defect Settlement Date",
    "Modification Flag",
    "Zero Balance Code",
    "Zero Balance Effective Date",
    "Current Interest Rate",
    "Current Deferred UPB",
    "Due Date of Last Paid Installment (DDLPI)",
    "MI Recoveries",
    "Net Sales Proceeds",
    "Non MI Recoveries",
    "Expenses",
    "Legal Costs",
    "Maintenance and Preservation Costs",
    "Taxes and Insurance",
    "Miscellaneous Expenses",
    "Actual Loss Calculation",
    "Modification Cost",
    "Step Modification Flag",
    "Deferred Payment Plan",
    "Estimated Loan-to-Value (ELTV)",
    "Zero Balance Removal UPB",
    "Delinquent Accrued Interest",
    "Delinquency Due to Disaster",
    "Borrower Assistance Status Code",
    "Current Month Modification Cost",
    "Interest Bearing UPB",
]
```

```
mortgages = pd.read_csv(
    "https://github.com/nickeubank/MIDS_Data/raw/master/mortgages/2004/"
    "sample_orig_2004_standard_mortgages.txt.zip?download=",
    sep="|",
    names=origination_colnames,
    compression="zip",
)

servicing = pd.read_csv(
    "https://github.com/nickeubank/MIDS_Data/raw/master/mortgages/2004/"
    "sample_svcg_2004_threeyears_standard_mortgages.txt.zip?download=",
    sep="|",
    names=service_colnames,
    compression="zip",
)
```

```
/var/folders/fs/h_8_rwsn5hvg9mhp0txgc_s9v6191b/T/ipykernel_21686/4057295142.py
  servicing = pd.read_csv(
```

```
print(f"2004 data has {len(mortgages):,.0f} new mortgages")
print(f"2004 data has {len(servicing):,.0f} servicing records")
```

```
2004 data has 17,471 new mortgages
2004 data has 379,461 servicing records
```

```
# Life becomes way easier if I add this here,
# though you may not realize it till
# around exercise 7
servicing = servicing[
    [
        "Monthly Reporting Period",
        "Current Loan Delinquency Status",
        "Loan Sequence Number",
    ]
]
```

Exercise 2

What is the unit of observation in `sample_orig_2004_standard_mortgages.txt` and in `sample_svcg_2004_threeyears_standard_mortgages.txt`?

> Each line of `orig` is one mortgages, while each entry of `svcg` is one mortgage *payment* (one per mortgage-month).

Exercise 3

Merge your two datasets. Be sure to use the `validate` keyword argument in `merge`.

You will find some records in the origination files not in the servicing file. We need data from both files, so just do an `inner` join.

Assuming that you list the data associated with `sample_orig_2004_standard_mortgages.txt` first and

`sample_svcg_2004_threeyears_standard_mortgages.txt` second, what keyword are you passing to `validate`? Store your answer as a string (use one of: `"1:1"`, `"m:1"`, `"1:m"`, `"m:m"`) in a dictionary called `results` under the key `ex2_merge_type`.

```
results = {}
results["ex2_merge_type"] = "1:m"
```

```
combined = pd.merge(
    mortgages,
    servicing,
    on="Loan Sequence Number",
    how="inner",
    validate="1:m",
    indicator=True,
)
combined._merge.value_counts()

assert (combined._merge == "both").all()
```

Exercise 4

For each unique mortgage in your dataset, create an indicator variable that takes on a value of 1 if, at any time during this period, the mortgage has been delinquent.

Delinquency status is stored in the variable `CURRENT LOAN DELINQUENCY STATUS`, and is coded as:

> CURRENT LOAN DELINQUENCY STATUS – A value corresponding to the number of days the borrower is delinquent, based on the due date of last paid installment (“DDLPI”) reported by servicers to Freddie Mac, and is calculated under the Mortgage Bankers Association (MBA) method. If a loan has been acquired by REO, then the Current Loan Delinquency Status will reflect the value corresponding to that status (instead of the value corresponding to the number of days the borrower is delinquent).
>
> 0 = Current, or less than 30 days delinquent
>
> 1 = 30-59 days delinquent
>
> 2=60–89days delinquent

> 3=90–119days delinquent
>
> And so on...
>
> RA = REO Acquisition

```
# Current Loan Delinquency Status
combined["Current Loan Delinquency Status"] = (
    combined["Current Loan Delinquency Status"].replace("RA", 99).astype("int"
)

combined["ever_delinquent"] = (
    combined.groupby("Loan Sequence Number")[
        "Current Loan Delinquency Status"
    ].transform("max")
    > 0
).astype("int")
```

Exercise 5

At this point, you should be able to drop all servicing variables reported on a monthly basis and just keep information about the original mortgage issuance (and still keep an indicator for whether the mortgage has ever been delinquent).

Store the final number of mortgages in your data under `ex5_num_mortgages` and the share (between 0 and 1) of mortgages that have been delinquent under `ex5_share_delinquent`. **Please round the share delinquent to 3 decimal places.**

```
combined.columns
```

```
Index(['Credit Score', 'First Payment Date', 'First Time Homebuyer Flag',
       'Maturity Date',
       'Metropolitan Statistical Area (MSA) Or Metropolitan Division',
       'Mortgage Insurance Percentage (MI %)', 'Number of Units',
       'Occupancy Status', 'Original Combined Loan-to-Value (CLTV)',
       'Original Debt-to-Income (DTI) Ratio', 'Original UPB',
       'Original Loan-to-Value (LTV)', 'Original Interest Rate', 'Channel',
       'Prepayment Penalty Mortgage (PPM) Flag',
       'Amortization Type (Formerly Product Type)', 'Property State',
       'Property Type', 'Postal Code', 'Loan Sequence Number', 'Loan Purpose',
       'Original Loan Term', 'Number of Borrowers', 'Seller Name',
       'Servicer Name', 'Super Conforming Flag',
       'Pre-HARP Loan Sequence Number', 'Program Indicator', 'HARP Indicator',
       'Property Valuation Method', 'Interest Only (I/O) Indicator',
       'Mortgage Insurance Cancellation Indicator', 'Monthly Reporting Period',
       'Current Loan Delinquency Status', '_merge', 'ever_delinquent'],
      dtype='object')
```

```
# This is made easier by drops earlier.
combined = combined.drop(
    columns=[
        "Current Loan Delinquency Status",
        "Monthly Reporting Period",
    ]
)
two_years = combined.drop_duplicates()
assert two_years["Loan Sequence Number"].is_unique
results["ex5_share_delinquent"] = np.round(two_years.ever_delinquent.mean(), 3
results["ex5_num_mortgages"] = len(two_years)

print(
    f"There are now {results['ex5_num_mortgages']:,.0f} unique mortgages in th
)
print(
    f"During the period studied, {results['ex5_share_delinquent']:.3%}"
    " of loans are delinquent at some point."
)
```

```
There are now 17,471 unique mortgages in the data.
During the period studied, 7.100% of loans are delinquent at some point.
```

Modeling Delinquency Risk

Your data should now be relatively tidy, in the technical sense of the term. And that means it should be relatively straightforward for you to build a model that answers the question “Given the features of a newly originated mortgage, how likely is the mortgage holder to fall into

delinquency within the first two years after origination?" to help your stakeholder decide which mortgages to purchase.

Exercise 6

For your analysis, include the following variables:

```
Credit Score
First Time Homebuyer Flag
Number of Units
Mortgage Insurance Percentage (MI %)
Occupancy Status
Original Debt-to-Income (DTI) Ratio
Original UPB
Original Loan-to-Value (LTV)
Original Interest Rate
Channel
Prepayment Penalty Mortgage (PPM) Flag
Amortization Type (Formerly Product Type)
Property State
Property Type
Original Loan Term
Number of Borrowers
```

Be sure to clean these variables. When doing so, please treat missing data as missing (e.g., `np.nan`, not as a distinct category). You will probably want to consult the documentation on the data for guidance on missing values.

```
```

```
# Clean up!
for i in [
    "Channel",
    "Property Valuation Method",
    "First Time Homebuyer Flag",
    "Occupancy Status",
]:
    two_years[i] = two_years[i].replace(9, np.nan)
    two_years[i] = two_years[i].replace("9", np.nan)

for i in ["Number of Units", "Property Type", "Number of Borrowers"]:
    two_years[i] = two_years[i].replace(99, np.nan)
    two_years[i] = two_years[i].replace("99", np.nan)

for i in [
    "Mortgage Insurance Percentage (MI %)",
    "Original Debt-to-Income (DTI) Ratio",
    "Original Loan-to-Value (LTV)",
]:
    two_years[i] = two_years[i].replace(999, np.nan)
    two_years[i] = two_years[i].replace("999", np.nan)

two_years["Credit Score"] = two_years["Credit Score"].replace(9999, np.nan)
two_years["Credit Score"] = two_years["Credit Score"].replace("9999", np.nan)
```

```
two_years["Number of Units"].value_counts()
```

```
Number of Units
1.0    17117
2.0      284
4.0       34
3.0       33
Name: count, dtype: int64
```

```
for_ml = two_years[
    [
        "Credit Score",
        "First Time Homebuyer Flag",
        "Number of Units",
        "Mortgage Insurance Percentage (MI %)",
        "Occupancy Status",
        "Original Debt-to-Income (DTI) Ratio",
        "Original UPB",
        "Original Loan-to-Value (LTV)",
        "Original Interest Rate",
        "Channel",
        "Prepayment Penalty Mortgage (PPM) Flag",
        "Amortization Type (Formerly Product Type)",
        "Property State",
        "Property Type",
        "Original Loan Term",
        "Number of Borrowers",
        "ever_delinquent",
        "Loan Sequence Number",
    ]
]
```

Why Aren't We Using Metropolitan Statistical Area (MSA)?

Metropolitan Statistical Area (MSA) is what the US Census Bureau calls what most people would think of as a city. Durham *plus* Chapel Hill are considered a single MSA, for example.

So looking at this list, you may be wondering why we aren't using the MSA variable in the data. After all, real estate is all about location, location, location, right?

Well, the problem is the data is too sparse — it's missing for a substantial number of observations, and more importantly there are lots of MSAs with only a couple of observations that are delinquent (about 40% of MSAs have less than 4 mortgages that are delinquent in their first two years). That's just too sparse for modelling — when coefficients are being estimated for those categories, they're bound to over-fit.

Put differently: would you be comfortable estimating the delinquency rate for, say, the city of Denver, CO with three or fewer delinquent observations? Of course not, but when you try and predict values to new data, that's precisely what you're doing — applying an estimate of delinquency from a few delinquent observations to any new observations in that MSA.

If you were working with longer time periods or the dataset with *all* Freddie Mac mortgages (not a sample), you could probably use MSA.

Exercise 7

The next step in our analysis is to convert our categorical variables to one-hot-encodings and use `train_test_split` to split our data.

To ensure replicability, **before** you `train_test_split` your data, please sort your data by `Loan Sequence Number`. This will ensure when we split the data with a random seed below, everyone will get the same split and the autograder will function.

You may create your one-hot-encodings however you wish, but I'm a fan of the patsy library's `dmatrices` function.

Hint: You should end up with 8 categorical variables, including some binary flags and `Number_of_Borrowers`, `Number_of_Units` (which you could argue should be continuous, but I think are better treated as categorical).

Hint 2: To use `patsy`, you will probably need to make some changes to the names of your variables. To make your life easier, you may wish to use a little snippet like this:

```
import re
mortgages_2004.columns = [re.sub(" ", "_", c) for c in mortgages_2004.columns]
mortgages_2004.columns = [re.sub("[%/()-]", "", c) for c in mortgages_2004.colu
```

Hint 3: your final `X` matrix should have 76 columns, including intercept.

Store the number of observations in your final dataset in `ex7_num_obs`.

```
import patsy
import re

# Patsy can't handle this type of punctuation in formulas
for_ml.columns = [re.sub(" ", "_", c) for c in for_ml.columns]
for_ml.columns = [re.sub("[%/()-]", "", c) for c in for_ml.columns]

for_ml.columns
```

```
Index(['Credit_Score', 'First_Time_Homebuyer_Flag', 'Number_of_Units',
       'Mortgage_Insurance_Percentage_MI_', 'Occupancy_Status',
       'Original_DebttoIncome_DTI_Ratio', 'Original_UPB',
       'Original_LoantoValue_LTV', 'Original_Interest_Rate', 'Channel',
       'Prepayment_Penalty_Mortgage_PPM_Flag',
       'Amortization_Type_Formerly_Product_Type', 'Property_State',
       'Property_Type', 'Original_Loan_Term', 'Number_of_Borrowers',
       'ever_delinquent', 'Loan_Sequence_Number'],
      dtype='object')
```

```
for_ml = for_ml.sort_values("Loan_Sequence_Number")

y, X = patsy.dmatrices(
    "ever_delinquent ~ Credit_Score"
    "+ First_Time_Homebuyer_Flag"
    "+ C(Number_of_Units)"
    "+ Mortgage_Insurance_Percentage_MI_"
    "+ Occupancy_Status"
    "+ Original_DebttoIncome_DTI_Ratio"
    "+ Original_UPB"
    "+ Original_LoantoValue_LTV"
    "+ Original_Interest_Rate"
    "+ Channel"
    "+ Prepayment_Penalty_Mortgage_PPM_Flag"
    "+ Amortization_Type_Formerly_Product_Type"
    "+ Property_State"
    "+ Property_Type"
    "+ Original_Loan_Term"
    "+ C(Number_of_Borrowers)",
    for_ml,
)
```

```
# Eyeball check result
X
```

```
DesignMatrix with shape (17052, 76)
  Columns:
    ['Intercept',
     'First_Time_Homebuyer_Flag[T.Y]',
     'C(Number_of_Units)[T.2.0]',
     'C(Number_of_Units)[T.3.0]',
     'C(Number_of_Units)[T.4.0]',
     'Occupancy_Status[T.P]',
     'Occupancy_Status[T.S]',
     'Channel[T.C]',
     'Channel[T.R]',
     'Channel[T.T]',
     'Prepayment_Penalty_Mortgage_PPM_Flag[T.Y]',
     'Property_State[T.AL]',
     'Property_State[T.AR]',
     'Property_State[T.AZ]',
     'Property_State[T.CA]',
     'Property_State[T.CO]',
     'Property_State[T.CT]',
     'Property_State[T.DC]',
     'Property_State[T.DE]',
     'Property_State[T.FL]',
     'Property_State[T.GA]',
     'Property_State[T.GU]',
     'Property_State[T.HI]',
     'Property_State[T.IA]',
     'Property_State[T.ID]',
     'Property_State[T.IL]',
     'Property_State[T.IN]',
     'Property_State[T.KS]',
     'Property_State[T.KY]',
     'Property_State[T.LA]',
     'Property_State[T.MA]',
     'Property_State[T.MD]',
     'Property_State[T.ME]',
     'Property_State[T.MI]',
     'Property_State[T.MN]',
     'Property_State[T.MO]',
     'Property_State[T.MS]',
     'Property_State[T.MT]',
     'Property_State[T.NC]',
     'Property_State[T.ND]',
     'Property_State[T.NE]',
     'Property_State[T.NH]',
     'Property_State[T.NJ]',
     'Property_State[T.NM]',
     'Property_State[T.NV]',
     'Property_State[T.NY]',
     'Property_State[T.OH]',
     'Property_State[T.OK]',
     'Property_State[T.OR]',
     'Property_State[T.PA]',
     'Property_State[T.PR]',
     'Property_State[T.RI]',
     'Property_State[T.SC]',
```

```
    'Property_State[T.SD]',
    'Property_State[T.TN]',
    'Property_State[T.TX]',
    'Property_State[T.UT]',
    'Property_State[T.VA]',
    'Property_State[T.VI]',
    'Property_State[T.VT]',
    'Property_State[T.WA]',
    'Property_State[T.WI]',
    'Property_State[T.WV]',
    'Property_State[T.WY]',
    'Property_Type[T.CP]',
    'Property_Type[T.MH]',
    'Property_Type[T.PU]',
    'Property_Type[T.SF]',
    'C(Number_of_Borrowers)[T.2.0]',
    'Credit_Score',
    'Mortgage_Insurance_Percentage_MI_',
    'Original_DebttoIncome_DTI_Ratio',
    'Original_UPB',
    'Original_LoantoValue_LTV',
    'Original_Interest_Rate',
    'Original_Loan_Term']
  Terms:
    'Intercept' (column 0)
    'First_Time_Homebuyer_Flag' (column 1)
    'C(Number_of_Units)' (columns 2:5)
    'Occupancy_Status' (columns 5:7)
    'Channel' (columns 7:10)
    'Prepayment_Penalty_Mortgage_PPM_Flag' (column 10)
    'Amortization_Type_Formerly_Product_Type' (columns 11:11)
    'Property_State' (columns 11:64)
    'Property_Type' (columns 64:68)
    'C(Number_of_Borrowers)' (column 68)
    'Credit_Score' (column 69)
    'Mortgage_Insurance_Percentage_MI_' (column 70)
    'Original_DebttoIncome_DTI_Ratio' (column 71)
    'Original_UPB' (column 72)
    'Original_LoantoValue_LTV' (column 73)
    'Original_Interest_Rate' (column 74)
    'Original_Loan_Term' (column 75)
  (to view full data, use np.asarray(this_obj))
```

```python
# Promised this in text above
assert X.shape[1] == 76

# Answer to Question
results["ex7_num_obs"] = X.shape[0]
print(f"The final dataset has {results['ex7_num_obs']:,.0f} observations")
```

```
The final dataset has 17,052 observations
```

Exercise 8

Use `train_test_split` from `sklearn.model_selection` to split the data.

Before you do, Use `0.2` as the `test_size` and use `random_state=42`.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Exercise 9

Now fit a `GradientBoostingClassifier` to the data (from `sklearn.ensemble`). Set `random_state=42` when instantiating your `GradientBoostingClassifier`.

```
from sklearn.ensemble import GradientBoostingClassifier

gb = GradientBoostingClassifier(random_state=42)
gb.fit(X_train, y_train.ravel())
```

```
▾  GradientBoostingClassifier
GradientBoostingClassifier(random_state=42)
```

Exercise 10

Use the `predict` method and your test data to generate a confusion matrix.

What problem do you see with the result? Round the share of observations predicted to be delinquent to three decimal places and store them under `ex10_predicted_delinquent` in `results`.

```

```

```python
# Predicts
y_pred = gb.predict(X_test)

# Confusion matrix
conf_matrix = pd.crosstab(
    y_test.squeeze(),
    y_pred.astype("int"),
    rownames=["Actually Delinquent"],
    colnames=["Predicted Delinquent"],
)

print(f"Confusion matrix:\n{conf_matrix}")
results["ex10_predicted_delinquent"] = np.round(
    conf_matrix.loc[:, 1].sum() / len(y_test), 3
)

print(
    "The share of values predicted delinquent (to three decimal places)"
    f" is {results['ex10_predicted_delinquent']:.3f}."
)
```

```
Confusion matrix:
Predicted Delinquent     0  1
Actually Delinquent
0.0                   3146  9
1.0                    252  4
The share of values predicted delinquent (to three decimal places) is 0.004.
```

> It's basically optimizing its objective function by just saying "no delinquency" (the dominant class), a canonical issue with unbalanced data.

Exercise 11

With imbalanced data, it's not uncommon for classification algorithms to maximize performance by basically just predicting everything is the dominant class.

One way to help reduce this behavior is to re-weight the data so that each observation of the dominant class gets less weight and each observation in the minority class gets more. A common default approach is to weigh each class with the reciprocal of its share of the data (so if delinquencies were 5% of our observations, we would give each delinquent observation a weight of 1/0.05 = 20, and each non-delinquent observation a weight of 1/0.95 = 1.0526...).

To accomplish this, create an array with sample weights using this reciprocal rule for each observation and pass it to the `sample_weight` argument of `.fit()` and refit your model.

To help the autograder, please:

- Recalculate the share of observations that are delinquent and use the full calculated value to calculate weights, and
- Re-instantiate your `GradientBoostingClassifier` with `random_state=42`.

What share of observations are now predicted to be delinquent? Store the share under `ex11_share_delinquent_weighted` in `results`. **Round your answer to three decimal places.**

```
delinquent = y_train.mean()
weights = np.where(y_train, 1 / delinquent, 1 / (1 - delinquent))
```

```
gb_weighted = GradientBoostingClassifier(random_state=42)
gb_weighted.fit(X_train, y_train.squeeze(), sample_weight=weights.squeeze())

# Predicts
y_pred_weighted = gb_weighted.predict(X_test)

# Score ane matrix
conf_matrix_weighted = pd.crosstab(
    y_test.squeeze(),
    y_pred_weighted.astype("int"),
    rownames=["Actually Delinquent"],
    colnames=["Predicted Delinquent"],
)
```

```
print(conf_matrix_weighted)

results["ex11_share_delinquent_weighted"] = np.round(
    conf_matrix_weighted.loc[:, 1].sum() / len(y_test), 3
)

print(
    "The share of values predicted delinquent (to three decimal places) with s
    f"weighting is now {results['ex11_share_delinquent_weighted']:.3f}."
)
```

```
Predicted Delinquent     0    1
Actually Delinquent
0.0                   2292  863
1.0                    102  154
The share of values predicted delinquent (to three decimal places) with sample
weighting is now 0.298.
```

Exercise 12

As you can tell, these weights are obviously parameters than can be tuned, but for the moment let's stick with the results of our model fit with reciprocal weights.

A classification model like this will provide a default classification threshold, but we can also use `.predict_proba()` to get *probabilities* the model assigns to whether each observation is a `1`. This is helpful, because one thing we can do as data scientists is play with the probability threshold at which we call an observation a `1`.

In other words, you can vary the threshold value `t` at which you say "for any observation with `.predict_proba()` above a value `t` is a 1, and any observation with a value below `t` is a 0." Then for each `t`, you will get a set of predicted classes for which you can calculate the resulting number of true and false positives and negatives.

(If you think in these terms, changing the sample weights we give the model will impact the orientation of the separating hyperplane the model creates to split the data; changing the classification threshold at which an observation is considered a `1` basically constitutes a simple shifting the hyperplane.)

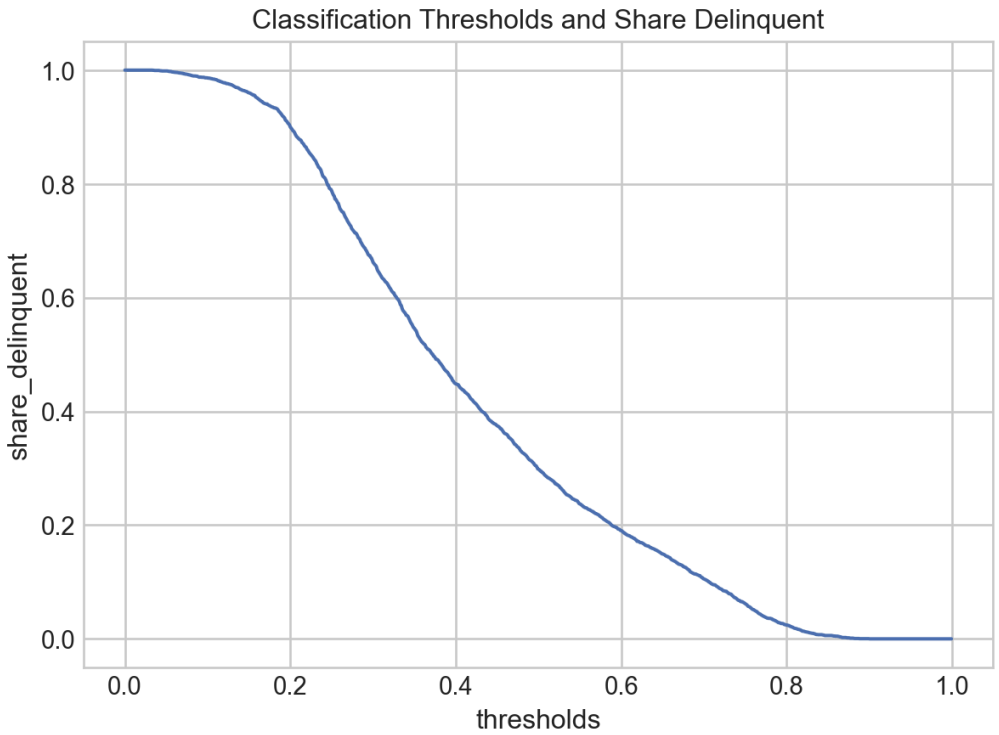
Using the results of our `GradientBoostingClassifier` fit with reciprocal sample weights, get the predicted probabilities for each observation in our test data.

Plot the share of observations classified as delinquent against classification thresholds from 0 to 1.

```
```

```
# Get proba
y_pred_proba_weighted = gb_weighted.predict_proba(X_test)[:, 1]

# Make a grid
grid = np.arange(0, 1, 0.001)
grid

# Make dataset to plot
to_plot = pd.DataFrame({"thresholds": grid})

to_plot["share_delinquent"] = to_plot["thresholds"].map(
    lambda threshold: (y_pred_proba_weighted > threshold).mean()
)
```

```
import seaborn.objects as so
from matplotlib import style
import matplotlib.pyplot as plt
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)

(
    so.Plot(to_plot, x="thresholds", y="share_delinquent")
    .add(so.Line())
    .label(title="Classification Thresholds and Share Delinquent")
    .theme({**style.library["seaborn-v0_8-whitegrid"]})
)
```



Exercise 13

From this visualization, you should be able to see that by changing our classification threshold, we can change the share of mortgages predicted to be delinquent (e.g., classified as dangerous to buy). If we *really* wanted to avoid risk, we could use a threshold like 0.2, and our client would basically not buy any mortgages. Or we could choose a threshold like 0.8 and our client would buy almost all the mortgages available!

So what threshold *should* we use?

Assume that, for your client, a good mortgage has a value of 1. (We can think of this as a normalization of any actual financial value). A delinquent mortgage the client has purchased has a value of -20 — that is, if the client bought 21 mortgages and 1 turned out to be delinquent, they would break even.

A good mortgage they fail to purchase they think is costing them about -0.05 times the value of a good mortgage. We can think of this as the "opportunity cost" of failing to buy a good mortgage and instead having to put their money somewhere else with a lower return.

This is the same as saying, when we think about our classification matrix, that the relative value of an observation in each cell is (normalized to the value of a True Negative (a safe mortgage predicted to be safe) being 1):

- True Positive: 0 (a mortgage correctly predicted to be delinquent our stakeholder didn't buy).
- True Negative: 1 (a mortgage correctly predicted to be safe our stakeholder did buy).
- False Negative: -20 (a mortgage incorrectly predicted to be safe our stakeholder did buy and that turned out to be delinquent).
- False Positive: -0.05 (a mortgage incorrectly predicted to be delinquent our stakeholder didn't buy but which turned out to be safe).

Again, without tuning our `sample_weights` (which you might do in a real analysis), what classification threshold would you choose?

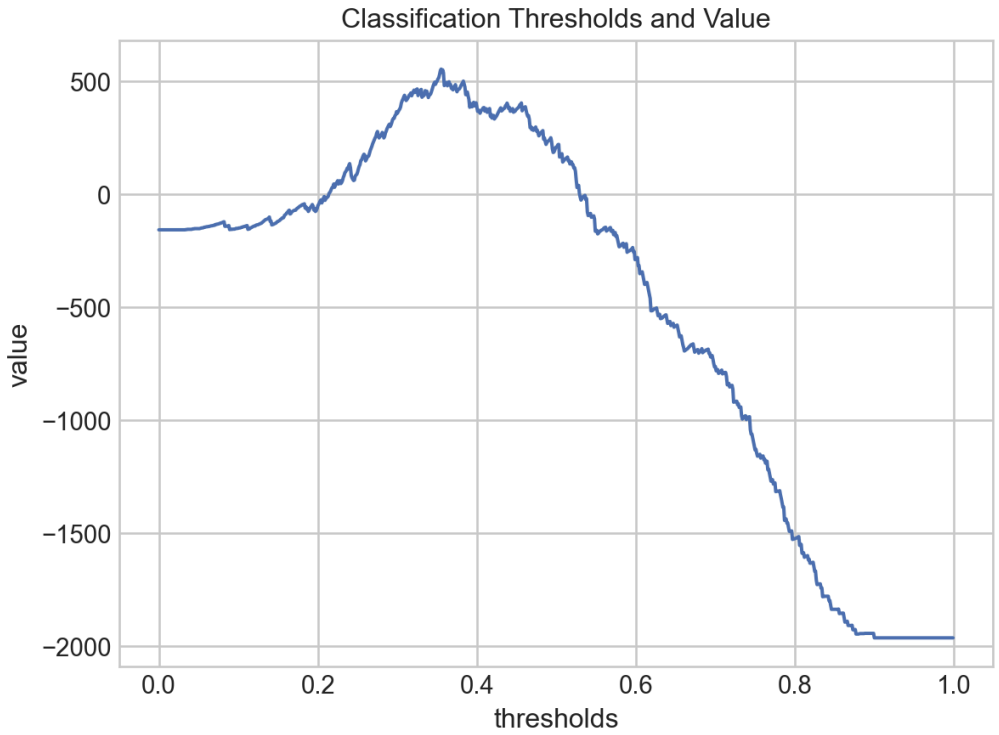
Do a grid search with ~1,000 grid steps. Find the threshold with the highest expected value and store it under the key `"ex13_optimal_threshold"`. **Round your answer to the nearest half tenth (0.05).**

I know, I know — this is a *very* weird rounding, but grid search is lumpy, so people will likely get slightly different answers, and rounding to two decimals isn't quite stable enough for an autograder. So your answer should be something like `0.15`, `0.20`, `0.25`, `0.30`, `0.35`, `0.40`, `0.45`, etc.

Please also plot your threshold against value calculations.

```
def value(threshold):

    TRUE_POS = 0
    TRUE_NEG = 1
    FALSE_NEG = -20
    FALSE_POS = -0.05

    confusion_matrix = pd.crosstab(
        y_test.squeeze(),
        (y_pred_proba_weighted > threshold).astype("int"),
        rownames=["Actually Delinquent"],
        colnames=["Predicted Delinquent"],
    )

    all_zeros = np.isclose((y_pred_proba_weighted > threshold).mean(), 0)
    all_ones = np.isclose((y_pred_proba_weighted > threshold).mean(), 1)

    if not all_zeros and not all_ones:
        utility = (
            TRUE_NEG * confusion_matrix.loc[0, 0]
            + (FALSE_NEG * confusion_matrix.loc[1, 0])
            + (FALSE_POS * confusion_matrix.loc[0, 1])
        )
    elif all_zeros:
        utility = (TRUE_NEG * confusion_matrix.loc[0, 0]) + (
            FALSE_NEG * confusion_matrix.loc[1, 0]
        )
    elif all_ones:
        utility = FALSE_POS * confusion_matrix.loc[0, 1]
    return utility


to_plot["value"] = to_plot["thresholds"].map(value)
```

```
(
    so.Plot(to_plot, x="thresholds", y="value")
    .add(so.Line())
    .label(title="Classification Thresholds and Value")
    .theme({**style.library["seaborn-v0_8-whitegrid"]})
)
```




```
optimal_threshold = to_plot.loc[
    to_plot["value"] == to_plot["value"].max(), "thresholds"
]

rounded_threshold = np.round(optimal_threshold.squeeze() * 20) / 20
```

```
results["ex13_optimal_threshold"] = rounded_threshold
print(
    f"The optimal threshold is about {results['ex13_optimal_threshold']:.3f} "
    "(plotted to three places to ensure I didn't screw up my rounding!)."
)
```

```
The optimal threshold is about 0.350 (plotted to three places to ensure I didn
```

Exercise 14

What is the value your customer will get at that (rounded-to-nearest-half-tenth) threshold per mortgage available (i.e., assume a True Negative yields a value of 1, calculate the value generated by your test data at your rounded optimal threshold, then normalize by the number of observations in the test data). Store this normalized value in `ex14_normalized_value`. **Round your answer to two decimal places.**

```
```

```
optimal_value = to_plot.loc[
    np.isclose(to_plot["thresholds"], results["ex13_optimal_threshold"]), "val
] / len(y_test)
optimal_value
```

```
350    0.146702
Name: value, dtype: float64
```

```
results["ex14_normalized_value"] = np.round(float(optimal_value), 2)
print(f"The normalized value at {results['ex14_normalized_value']:.2f}.")
```

```
The normalized value at 0.15.
```

Now To The Future

Most of the time, at least as students, we never get to see how well our predictions do. We fit our models on our training data, then evaluate their performance against our test data, and because we're evaluating our model against data it hadn't seen during training, we act as if we're really evaluating how well our predictions would fair if deployed in the real world.

But our mortgage data is from 2004, and as you may have noticed, 2004 was actually a while ago, which means we can now see how the model we trained on the first 24 months of payments on mortgages that originated in 2004 would do if we actually deployed it. To have our 24 months of payments, of course, the soonest we could have done the analysis above would have been in late 2006. So let's assume that your boss *immediately* deploys your model and starts buying up all the mortgages your model says should be purchased starting on January 1st 2007 and continued through December 31st 2007.

Would you have gotten the average value per mortgage your model predicted?

Exercise 15

In this folder you will find data on mortgages originated in 2007 along with servicing data from 2007, 2008, and 2009.

Please:

- load this data (again, from a URL to help the autograder).
- clean up your data as you did the data from the earlier period,
- use patsy to prepare the data so you can use the model **from above** (the model that used the reciprocal weights on data from mortgages originating in 2004) with this data.

In other words, we're using the model you trained on 2004 data to predict credit risk for these new mortgages. We're doing this to simulate what you would do if you were to actually put the model you fit in Exercise 11 "into production" to model the risk of new mortgages. So you won't use the `.fit()` method again, just the `.predict()` method with the new data.

Warning: Because you are asking sklearn to use a model trained on one dataset to predict values using predictors from a second dataset, any difference in the structure of the second dataset will cause problems. For example, if `num_of_units` is an integer in the 2007 data when you try and run your predictions, but it was a float in the 2004 data, patsy will give the columns slightly different names and you'll get an error like this:

```
ValueError: The feature names should match those that were passed during fit.
Feature names unseen at fit time:
- C(num_of_units)[T.2]
- C(num_of_units)[T.3]
- C(num_of_units)[T.4]
Feature names seen at fit time, yet now missing:
- C(num_of_units)[T.2.0]
- C(num_of_units)[T.3.0]
- C(num_of_units)[T.4.0]
```

This can be corrected by ensuring that `num_of_units` is of the same type when you run `dmatrices` for both datasets.

Your final X matrix should still have 76 columns. As sanity checks, store the final number of observations in your data after using patsy to make a design matrix in `"ex15_num_obs"` and the share of mortgages in your design matrix that are actually delinquent in `"ex15_share_delinquent"`.

Round the share delinquent to three decimal places.

Be sure you calculate the share delinquent *in your design matrix* — some observations get dropped when you use patsy because some model feature observations have missing values, so the answer will be slightly different than the value for the data that went into patsy.

Hint: The 2007 delinquency rate should be higher than the 2004 delinquency rate for reasons you can probably figure out if you google it. Something happened with mortgages between 2007 and 2009...

```
###########
# Load
###########

mortgages_2007 = pd.read_csv(
    "https://github.com/nickeubank/MIDS_Data/raw/master/mortgages/2007/"
    "sample_orig_2007_standard_mortgages.txt.zip?download=",
    sep="|",
    names=origination_colnames,
    compression="zip",
)

servicing_2007 = pd.read_csv(
    "https://github.com/nickeubank/MIDS_Data/raw/master/mortgages/2007/"
    "sample_svcg_2007_threeyears_standard_mortgages.txt.zip?download=",
    sep="|",
    names=service_colnames,
    compression="zip",
)

print(f"2007 data has {len(mortgages_2007):,.0f} new mortgages")
print(f"2007 data has {len(servicing_2007):,.0f} servicing records")
```

```
2007 data has 22,542 new mortgages
2007 data has 481,871 servicing records
```

```
/var/folders/fs/h_8_rwsn5hvg9mhp0txgc_s9v6191b/T/ipykernel_21686/1217561400.py
  servicing_2007 = pd.read_csv(
```

```
########
# NOTE: In any serious real workflow,
# the way to do this would be to write a .py script
# and parameterize it, not copy-paste the code.
# This kind of duplication invites problems if, for
# example, you change code above and forget to change it
# down here as well.
########

# Life becomes way easier if I add this here,
servicing_2007 = servicing_2007[
    [
        "Current Loan Delinquency Status",
        "Loan Sequence Number",
    ]
]

combined_2007 = pd.merge(
    mortgages_2007,
    servicing_2007,
    on="Loan Sequence Number",
    how="inner",
    validate="1:m",
    indicator=True,
)
combined_2007._merge.value_counts()
assert (combined_2007._merge == "both").all()
```

```
# Current Loan Delinquency Status
combined_2007["Current Loan Delinquency Status"] = (
    combined_2007["Current Loan Delinquency Status"].replace("RA", 99).astype(
)

combined_2007["ever_delinquent"] = (
    combined_2007.groupby("Loan Sequence Number")[
        "Current Loan Delinquency Status"
    ].transform("max")
    > 0
).astype("int")
```

```
```

```
# Clean missings
for i in [
    "Channel",
    "Property Valuation Method",
    "First Time Homebuyer Flag",
    "Occupancy Status",
]:
    combined_2007[i] = combined_2007[i].replace(9, np.nan)
    combined_2007[i] = combined_2007[i].replace("9", np.nan)

for i in ["Number of Units", "Property Type", "Number of Borrowers"]:
    combined_2007[i] = combined_2007[i].replace(99, np.nan)
    combined_2007[i] = combined_2007[i].replace("99", np.nan)


for i in [
    "Mortgage Insurance Percentage (MI %)",
    "Original Debt-to-Income (DTI) Ratio",
    "Original Loan-to-Value (LTV)",
]:
    combined_2007[i] = combined_2007[i].replace(999, np.nan)
    combined_2007[i] = combined_2007[i].replace("999", np.nan)

combined_2007["Credit Score"] = combined_2007["Credit Score"].replace(9999, np
combined_2007["Credit Score"] = combined_2007["Credit Score"].replace("9999",
```

```
# Subset columns
for_ml_2007 = combined_2007[
    [
        "Credit Score",
        "First Time Homebuyer Flag",
        "Number of Units",
        "Mortgage Insurance Percentage (MI %)",
        "Occupancy Status",
        "Original Debt-to-Income (DTI) Ratio",
        "Original UPB",
        "Original Loan-to-Value (LTV)",
        "Original Interest Rate",
        "Channel",
        "Prepayment Penalty Mortgage (PPM) Flag",
        "Amortization Type (Formerly Product Type)",
        "Property State",
        "Property Type",
        "Original Loan Term",
        "Number of Borrowers",
        "ever_delinquent",
        "Loan Sequence Number",
    ]
]
```

```
```

```
for_ml_2007 = for_ml_2007.drop_duplicates()
assert two_years["Loan Sequence Number"].is_unique
```

```
# Patsy can't handle this type of punctuation in formulas
for_ml_2007.columns = [re.sub(" ", "_", c) for c in for_ml_2007.columns]
for_ml_2007.columns = [re.sub("[%/()-]", "", c) for c in for_ml_2007.columns]

y_2007, X_2007 = patsy.dmatrices(
    "ever_delinquent ~ Credit_Score"
    "+ First_Time_Homebuyer_Flag"
    "+ C(Number_of_Units)"
    "+ Mortgage_Insurance_Percentage_MI_"
    "+ Occupancy_Status"
    "+ Original_DebttoIncome_DTI_Ratio"
    "+ Original_UPB"
    "+ Original_LoantoValue_LTV"
    "+ Original_Interest_Rate"
    "+ Channel"
    "+ Prepayment_Penalty_Mortgage_PPM_Flag"
    "+ Amortization_Type_Formerly_Product_Type"
    "+ Property_State"
    "+ Property_Type"
    "+ Original_Loan_Term"
    "+ C(Number_of_Borrowers)",
    for_ml_2007,
)
```

```
assert X_2007.shape[1] == 76
```

```
results["ex15_num_obs"] = X_2007.shape[0]
results["ex15_share_delinquent"] = float(np.round(y_2007.mean(), 3))
print(f"Num obs in the 2007 data is {results['ex15_num_obs']:,.0f}")
print(
    f"share of these 2007 morgages that are delinquent is {results['ex15_share
)
```

```
Num obs in the 2007 data is 21,972
share of these 2007 morgages that are delinquent is 11.0%
```

Exercise 16

Had your stakeholder purchased all the mortgages originating in 2007 using your model trained on 2004 mortgages, what would the average normalized value of those mortgages

be?

Store your result under the key `"ex16_normalized_value_2007"`. **Round your answer to 2 decimal places.**

Calculate the actual return your model provided as a percentage of the predicted return (e.g., `100 * results["ex16_normalized_value_2007"] / results["ex14_normalized_value"]`, so 1 is one percent, 100 is 100 percent).

To be clear, you should do this calculation with the *rounded* values you stored in `results` (again, for the autograder — in general you should never round until the end of calculations, but we're trying to smooth little differences).

Store this result as `"ex16_final_return_pct"`. **Round this calculated percentage one decimal place.**

```
# Predicts
y_2007_pred_proba = gb_weighted.predict_proba(X_2007)[:, 1]

y_2007_predicted = (y_2007_pred_proba > results["ex13_optimal_threshold"]).ast

TRUE_POS = 0
TRUE_NEG = 1
FALSE_NEG = -20
FALSE_POS = -0.05

confusion_matrix_2007 = pd.crosstab(
    y_2007.squeeze(),
    y_2007_predicted,
    rownames=["Actually Delinquent"],
    colnames=["Predicted Delinquent"],
)

normalized_value_2007 = (
    TRUE_NEG * confusion_matrix_2007.loc[0, 0]
    + (FALSE_NEG * confusion_matrix_2007.loc[1, 0])
    + (FALSE_POS * confusion_matrix_2007.loc[0, 1])
) / len(y_2007)

results["ex16_normalized_value_2007"] = np.round(normalized_value_2007, 2)
print(f"Realized value would be {results['ex16_normalized_value_2007']:.3f}")
```

```
Realized value would be 0.030
```

```
results["ex16_final_return_pct"] = np.round(
    (results["ex16_normalized_value_2007"] / results["ex14_normalized_value"])
)
print(
    f"The model returned {results['ex16_final_return_pct']:.1f}% of what was p
)
```

```
The model returned 20.0% of what was predicted
```

Exercise 17

How did the performance of your model against your test data (from 2004) compare to your model's actual performance in later years (the 2007 data)? What lesson from our class readings does this illustrate, and how does it relate to internal and external validity?

The question is worth several points, and is meant to be the place where you reflect on why I made you do all this. Your answer should not be a couple sentences.

> The later performance was much lower. test-train-split only tells you about accuracy you can expect *internally*; it doesn't account for *external validity* induced uncertainty.

```
sorted(set(results.keys()))
```

```
['ex10_predicted_delinquent',
 'ex11_share_delinquent_weighted',
 'ex13_optimal_threshold',
 'ex14_normalized_value',
 'ex15_num_obs',
 'ex15_share_delinquent',
 'ex16_final_return_pct',
 'ex16_normalized_value_2007',
 'ex2_merge_type',
 'ex5_num_mortgages',
 'ex5_share_delinquent',
 'ex7_num_obs']
```

```
len(results.keys())
```

```
12
```

```
assert set(results.keys()) == {
    "ex2_merge_type",
    "ex5_num_mortgages",
    "ex5_share_delinquent",
    "ex7_num_obs",
    "ex10_predicted_delinquent",
    "ex11_share_delinquent_weighted",
    "ex13_optimal_threshold",
    "ex14_normalized_value",
    "ex15_num_obs",
    "ex15_share_delinquent",
    "ex16_final_return_pct",
    "ex16_normalized_value_2007",
}
```