CAPSTONE PROJECT

MACHINE LEARNING ENGINEERING NANODEGREE

Name: **Fernando Damasio** E-mail: **fernando@cashflix.com.br** Institution: **Udacity**

CONTENTS

1. DEFINITION

1.1. Project Overview

The problem and the data sets was taken from Kaggle[1], a website that runs programming contests to crowdsource machine learning solutions. The competition chosen was the Allstate Claims Severity[2].

Allstate[3] is the largest publicly held personal lines property and casualty insurer in America, serving more than 16 million households nationwide.

Next, information about the contest:

- *Started:* 6:02 pm, Monday 10 October 2016 UTC
- *Ends:* 11:59 pm, Monday 12 December 2016 UTC (63 total days)

The given data sets have 3 files:

1. *train.csv:* The training set.
2. *test.csv:* The test set.
3. *sample_submission.csv:* A sample submission file in the correct format.

For this project, the following AWS instance[4] will be used to train the model.

Infrastructure

- VSH: Amazon EC2
- Availability zone: us-east-1d

[1] https://www.kaggle.com/
[2] https://www.kaggle.com/c/allstate-claims-severity
[3] https://www.allstate.com/
[4] https://aws.amazon.com/

- Instance type: C3
- Application: Compute Optimized
- Model: c3.2xlarge
- vCPU: 8
- Mem (GiB): 15
- SSD Storage (GB): 2x80
- Operational system: Windows Server 2012 R2
- Platform: Anaconda2 64 bits
- Environment: Jupyter Notebook

Features:

- High Frequency Intel Xeon E5-2680 v2 (Ivy Bridge) Processors
- Support for Enhanced Networking
- Support for clustering
- SSD-backed instance storage

1.2. Problem Statement

Each row in this dataset represents an insurance claim. Variables prefaced with 'cat' are categorical, while those prefaced with 'cont' are continuous. The 'loss' value must the predicted for each id in the test set. It's a Supervised Learning type of problem.

The tasks involved are the following:

1. Download the data set.
2. Define the computation infrastructure necessary to process the data.
3. Visualize the train and test data sets.
4. Preprocess continuous features.
5. Preprocess categorical features.
6. Preprocess the target column.
7. Split the train and test data sets (as the test set of the contest is only for submitting purpose).
8. Define the best model for the problem.
9. Tune the best model.
10. Predict 'loss' column for the test set.
11. Postprocess the 'loss' column predicted.
12. Present the data as the sample_submission.csv file.

The final model is expected to predict with accuracy the loss for any new insurance claim with the same features as input.

1.3. Metrics

The metric used is the Mean Absolute Error as it is the score defined for the leaderboard table[5] for the Kaggle contest.

In statistics, the mean absolute error (MAE) is a quantity used to measure how close forecasts or predictions are to the eventual outcomes. The mean absolute error is given by:

$$MAE = \frac{1}{n}\sum_{t=1}^{n}|x_t - x'_t|$$

Where n is the number of predictions, x_t is the t*th* true value and x'_t is the t*th* predicted value.

2. ANALYSIS

2.1. Data Exploration

First, the data must be imported defining the column 'id' as the index column for the pandas DataFrame. And then, the number of rows and columns for each data set is shown:

2.1.1. Load CSV

```
import pandas as pd
import numpy as np
from time import time

start = time()

train = pd.read_csv('Z:/data/allstate/train.csv', index_col='id')
test = pd.read_csv('Z:/data/allstate/test.csv', index_col='id')

end = time()
print "Load CSVs in {:.1f} seconds.\n".format(end - start)
```

```
Load CSVs in 4.4 seconds.
```

```
print train.shape
print test.shape
```

```
(188318, 131)
(125546, 130)
```

After printing the data shape, we discover that the train data set has 188,318 rows and 131 columns and the test data set has 125.546 rows and 130 columns.

Now the columns of each data set are presented:

[5] https://www.kaggle.com/c/allstate-claims-severity/leaderboard

2.1.2. Print Columns

```
print list(train.columns)
print ""
print list(test.columns)
```

```
['cat1', 'cat2', 'cat3', 'cat4', 'cat5', 'cat6', 'cat7', 'cat8', 'cat9',
'cat10', 'cat11', 'cat12', 'cat13', 'cat14', 'cat15', 'cat16', 'cat17',
'cat18', 'cat19', 'cat20', 'cat21', 'cat22', 'cat23', 'cat24', 'cat25',
'cat26', 'cat27', 'cat28', 'cat29', 'cat30', 'cat31', 'cat32', 'cat33',
'cat34', 'cat35', 'cat36', 'cat37', 'cat38', 'cat39', 'cat40', 'cat41',
'cat42', 'cat43', 'cat44', 'cat45', 'cat46', 'cat47', 'cat48', 'cat49',
'cat50', 'cat51', 'cat52', 'cat53', 'cat54', 'cat55', 'cat56', 'cat57',
'cat58', 'cat59', 'cat60', 'cat61', 'cat62', 'cat63', 'cat64', 'cat65',
'cat66', 'cat67', 'cat68', 'cat69', 'cat70', 'cat71', 'cat72', 'cat73',
'cat74', 'cat75', 'cat76', 'cat77', 'cat78', 'cat79', 'cat80', 'cat81',
'cat82', 'cat83', 'cat84', 'cat85', 'cat86', 'cat87', 'cat88', 'cat89',
'cat90', 'cat91', 'cat92', 'cat93', 'cat94', 'cat95', 'cat96', 'cat97',
'cat98', 'cat99', 'cat100', 'cat101', 'cat102', 'cat103', 'cat104',
'cat105', 'cat106', 'cat107', 'cat108', 'cat109', 'cat110', 'cat111',
'cat112', 'cat113', 'cat114', 'cat115', 'cat116', 'cont1', 'cont2',
'cont3', 'cont4', 'cont5', 'cont6', 'cont7', 'cont8', 'cont9', 'cont10',
'cont11', 'cont12', 'cont13', 'cont14', 'loss']

['cat1', 'cat2', 'cat3', 'cat4', 'cat5', 'cat6', 'cat7', 'cat8', 'cat9',
'cat10', 'cat11', 'cat12', 'cat13', 'cat14', 'cat15', 'cat16', 'cat17',
'cat18', 'cat19', 'cat20', 'cat21', 'cat22', 'cat23', 'cat24', 'cat25',
'cat26', 'cat27', 'cat28', 'cat29', 'cat30', 'cat31', 'cat32', 'cat33',
'cat34', 'cat35', 'cat36', 'cat37', 'cat38', 'cat39', 'cat40', 'cat41',
'cat42', 'cat43', 'cat44', 'cat45', 'cat46', 'cat47', 'cat48', 'cat49',
'cat50', 'cat51', 'cat52', 'cat53', 'cat54', 'cat55', 'cat56', 'cat57',
'cat58', 'cat59', 'cat60', 'cat61', 'cat62', 'cat63', 'cat64', 'cat65',
'cat66', 'cat67', 'cat68', 'cat69', 'cat70', 'cat71', 'cat72', 'cat73',
'cat74', 'cat75', 'cat76', 'cat77', 'cat78', 'cat79', 'cat80', 'cat81',
'cat82', 'cat83', 'cat84', 'cat85', 'cat86', 'cat87', 'cat88', 'cat89',
'cat90', 'cat91', 'cat92', 'cat93', 'cat94', 'cat95', 'cat96', 'cat97',
'cat98', 'cat99', 'cat100', 'cat101', 'cat102', 'cat103', 'cat104',
'cat105', 'cat106', 'cat107', 'cat108', 'cat109', 'cat110', 'cat111',
'cat112', 'cat113', 'cat114', 'cat115', 'cat116', 'cont1', 'cont2',
'cont3', 'cont4', 'cont5', 'cont6', 'cont7', 'cont8', 'cont9', 'cont10',
'cont11', 'cont12', 'cont13', 'cont14']
```

Both data sets have 116 categorical features and 16 numerical features. The ‘loss’ column is only presented in the train data set as it is the target variable to learn the model.

Now we define new variables to work with based on the data sets.

2.1.3. Define Variables

```
loss = train['loss']
features = train.drop('loss', 1)
train_test = pd.concat((features, test))
numeric_features = list(train_test.dtypes[train_test.dtypes !=
"object"].index)
categorical_features = list(features.drop(numeric_features, 1))
```

The train_test variable above is important to measure, visualize and transform the train and test data together.

To better understand the numerical features, the following parameters are calculated: count, mean, std, min, 25%, 50%, 75%, max.

2.1.4. Describe Data

```
print "Train_Test Data Set Describe"
print train_test.describe()
print "\nLoss Describe"
print loss.describe()
```

```
Train_Test Data Set Describe
                cont1           cont2           cont3           cont4  \
count  313864.000000  313864.000000  313864.000000  313864.000000
mean        0.494096       0.507089       0.498653       0.492021
std         0.187768       0.207056       0.201961       0.211101
min         0.000016       0.001149       0.002634       0.176921
25%         0.347403       0.358319       0.336963       0.327354
50%         0.475784       0.555782       0.527991       0.452887
75%         0.625272       0.681761       0.634224       0.652072
max         0.984975       0.862654       0.944251       0.956046

                cont5           cont6           cont7           cont8  \
count  313864.000000  313864.000000  313864.000000  313864.000000
mean        0.487513       0.491442       0.485360       0.486823
std         0.209063       0.205394       0.178531       0.199442
min         0.281143       0.012683       0.069503       0.236880
25%         0.281143       0.336105       0.351299       0.317960
50%         0.422268       0.440945       0.438650       0.441060
75%         0.643315       0.655818       0.591165       0.623580
max         0.983674       0.997162       1.000000       0.982800

               cont9          cont10          cont11          cont12  \
count  313864.00000  313864.000000  313864.000000  313864.000000
mean        0.48571       0.498403       0.493850       0.493503
std         0.18185       0.185906       0.210002       0.209716
min         0.00008       0.000000       0.035321       0.036232
25%         0.35897       0.364580       0.310961       0.314945
50%         0.44145       0.461190       0.457203       0.462286
75%         0.56889       0.619840       0.678924       0.679096
max         0.99540       0.994980       0.998742       0.998484

               cont13          cont14
count  313864.000000  313864.000000
mean        0.493917       0.495665
std         0.212911       0.222537
min         0.000228       0.178568
25%         0.315758       0.294657
50%         0.363547       0.407020
75%         0.689974       0.724707
max         0.988494       0.844848

Loss Describe
count    188318.000000
mean       3037.337686
std        2904.086186
min           0.670000
25%        1204.460000
50%        2115.570000
75%        3864.045000
max      121012.250000
```

```
Name: loss, dtype: float64
```

All the numerical features values are between 0 and 1. We can presume that the values were already scaled and transformed to be presented to the competitors of the contest. The target variable 'loss' values are between 0.67 and 121,012.25.

Another parameter to better understand the numerical data is the Skew.

2.1.5. Print Skew

```
from scipy.stats import skew

print "Train_Test Data Set Skew"
print train_test.skew()
print "\nLoss Skew"
print loss.skew()
```

```
Train_Test Data Set Skew
cont1     0.513207
cont2    -0.311147
cont3    -0.007023
cont4     0.417561
cont5     0.679614
cont6     0.458415
cont7     0.825893
cont8     0.673240
cont9     1.067252
cont10    0.352118
cont11    0.281141
cont12    0.291998
cont13    0.376140
cont14    0.250674
dtype: float64

Loss Skew
3.79495837754
```

All columns, including 'loss' target variable area skewed, except cont 3. This is a particularly important analysis for preprocessing the data for linear models.

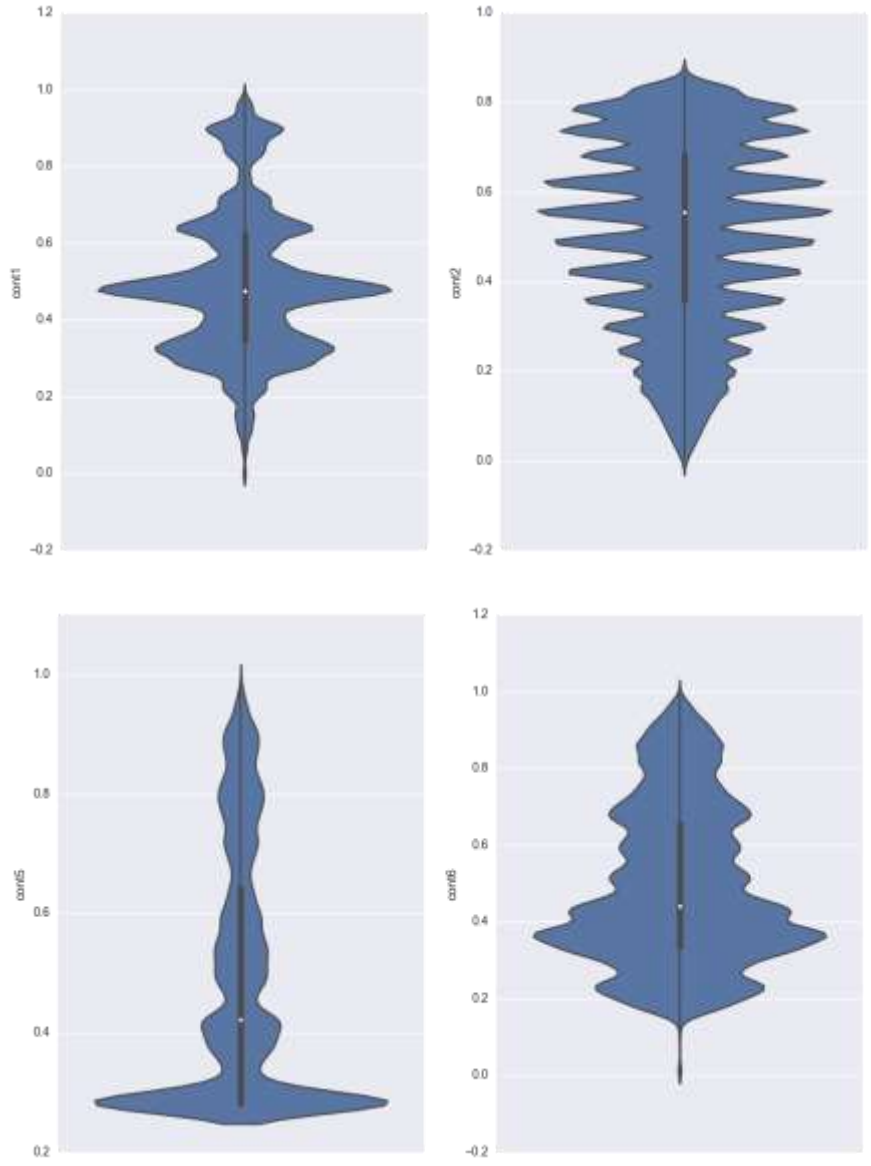
2.2. Exploratory Visualization

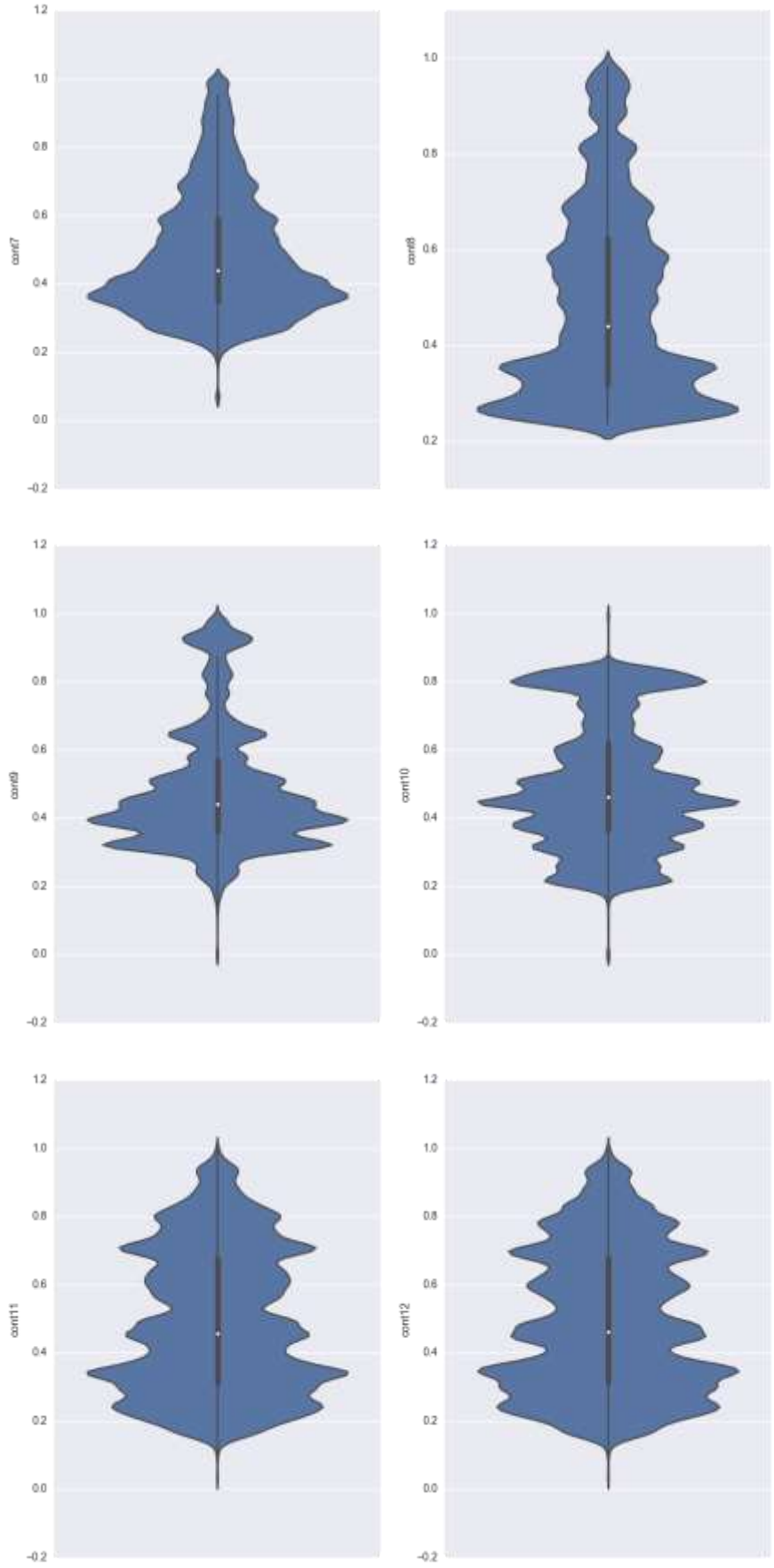
A different kind of plot is better to visualize each type of data. For the numerical values, the Violin Plot will be used in order to see the distribution. For the categorical values, the count plot will be used.

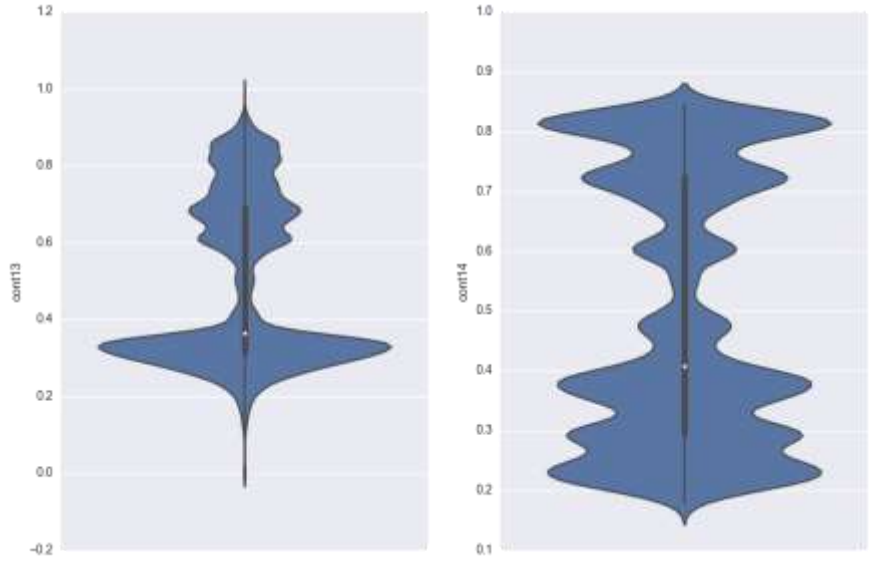
2.2.1. Numerical Features

```
# Import plotting libraries
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# Create a dataframe with only numerical features
data = train_test[numeric_features]
```

```
# Get the names of the columns
cols=data.columns

# Plot violin for all attributes in a 7x2 grid
n_cols = 2
n_rows = 7

for i in range(n_rows):
    fg,ax = plt.subplots(nrows=1,ncols=n_cols,figsize=(12, 8))
    for j in range(n_cols):
        sns.violinplot(y=cols[i*n_cols+j], data=data, ax=ax[j])
```

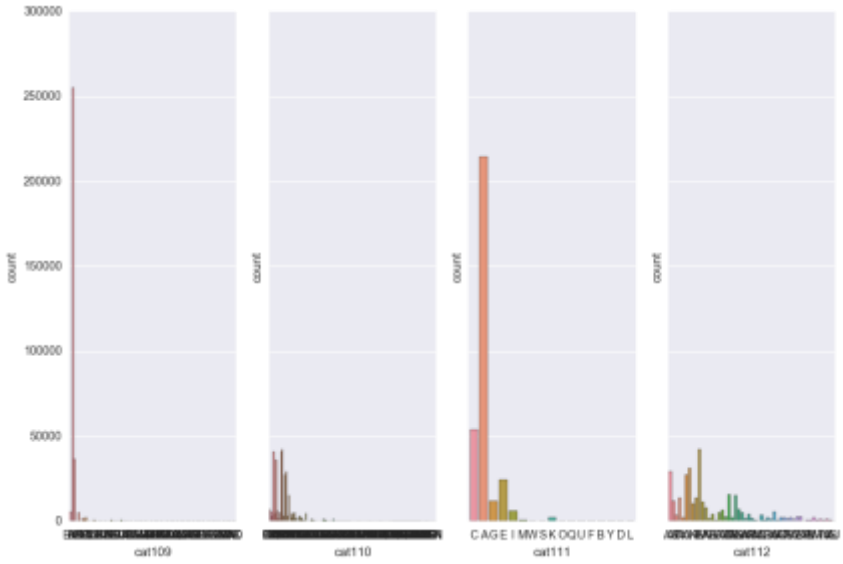


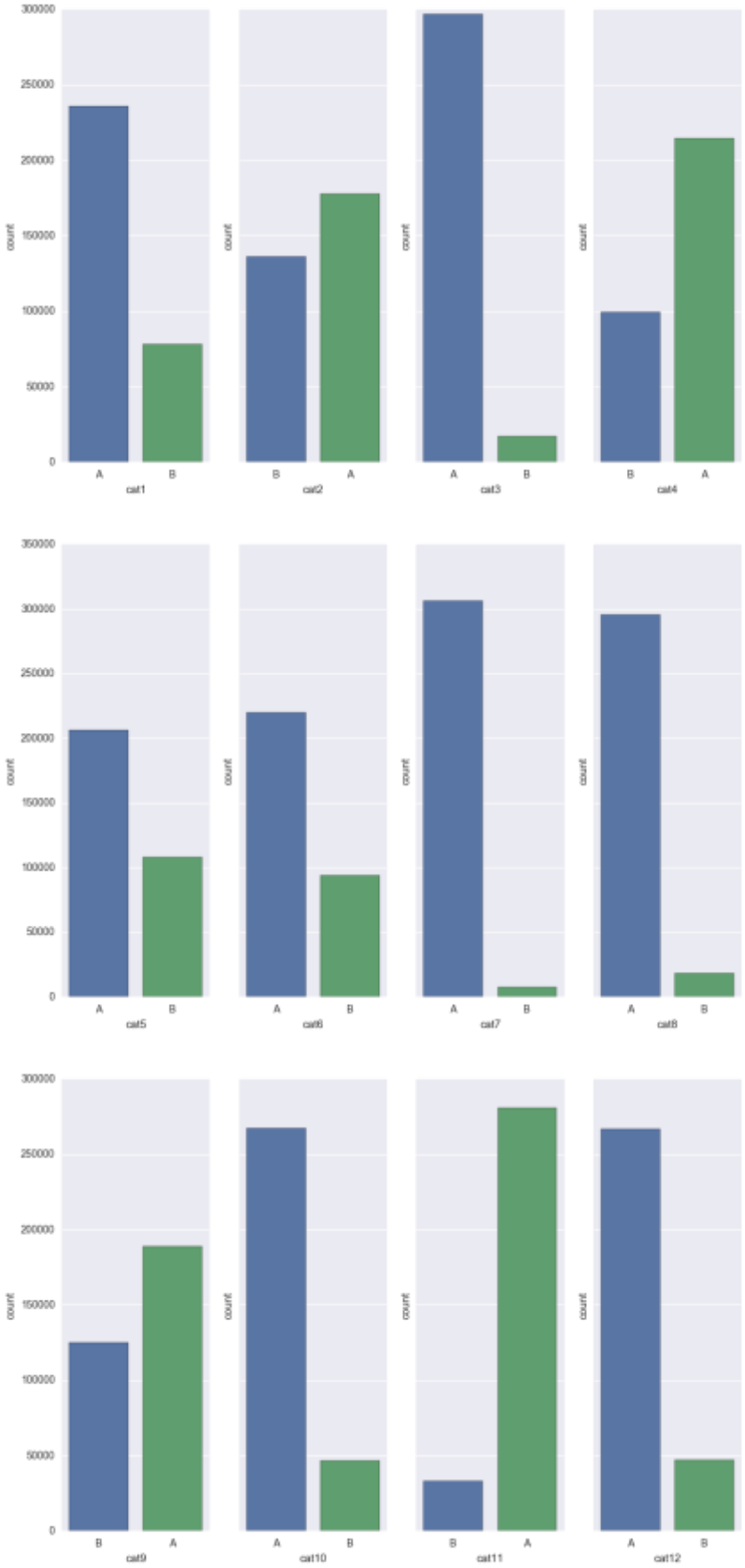

conf7
conf8
conf9
conf10
conf11
conf12

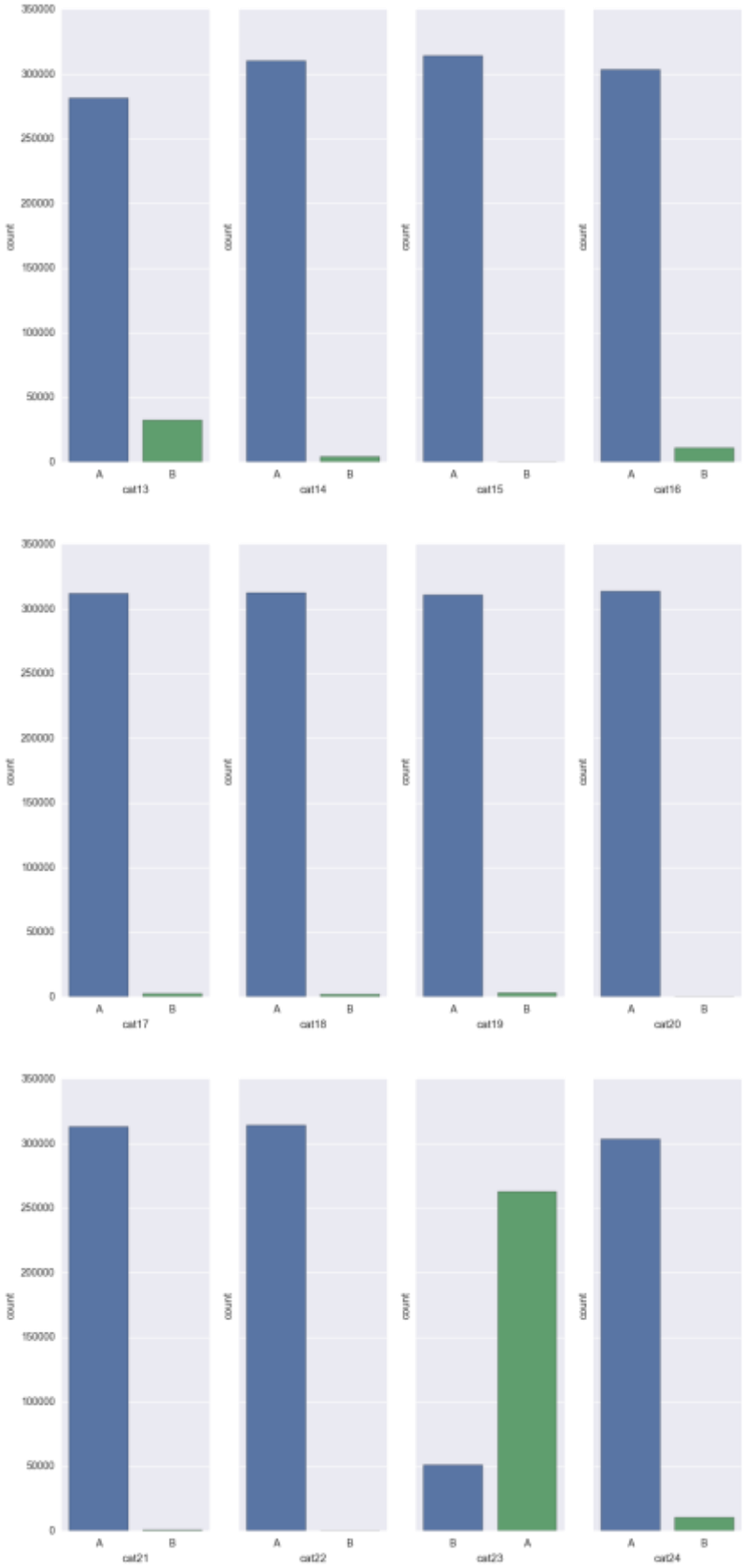


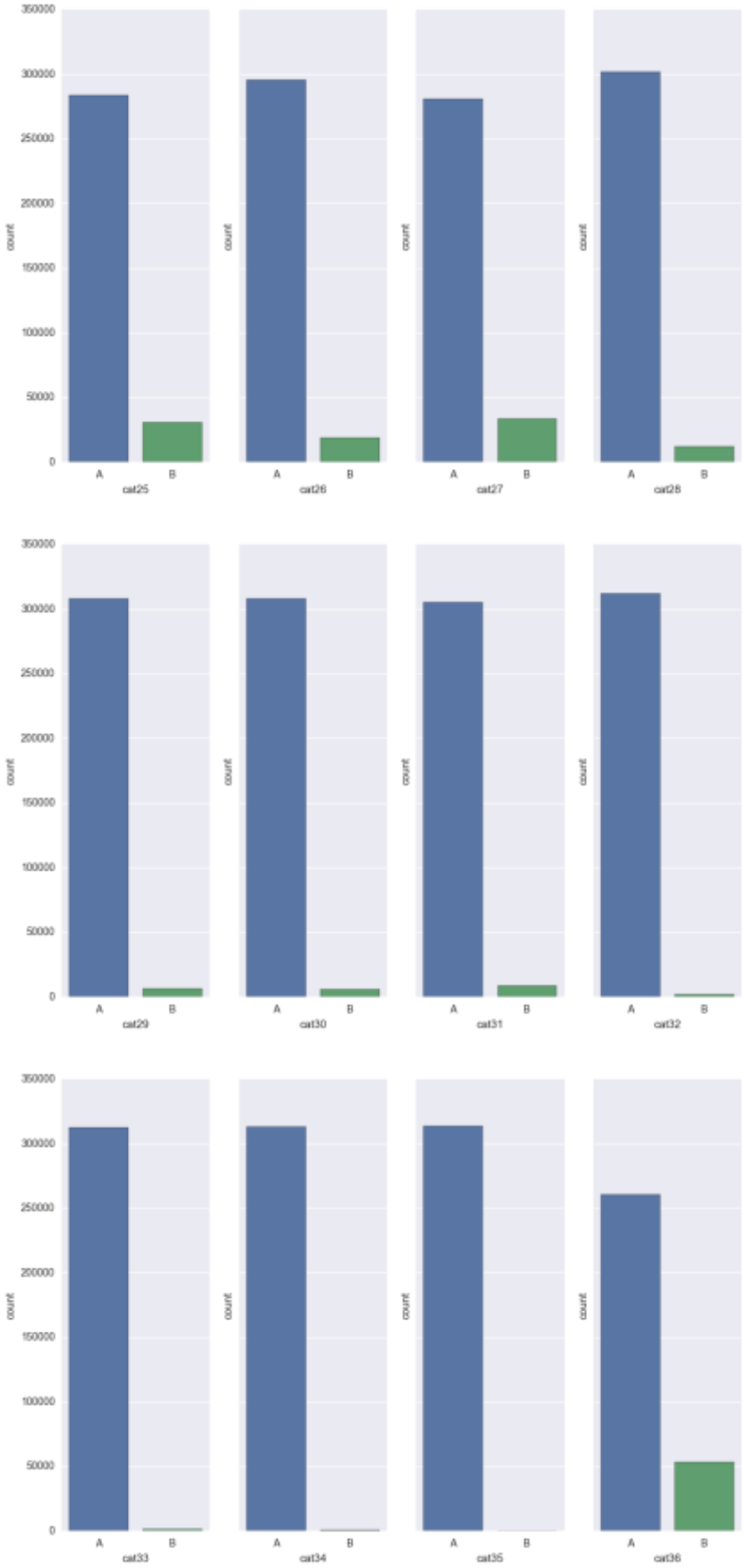
2.2.2. Categorical Features

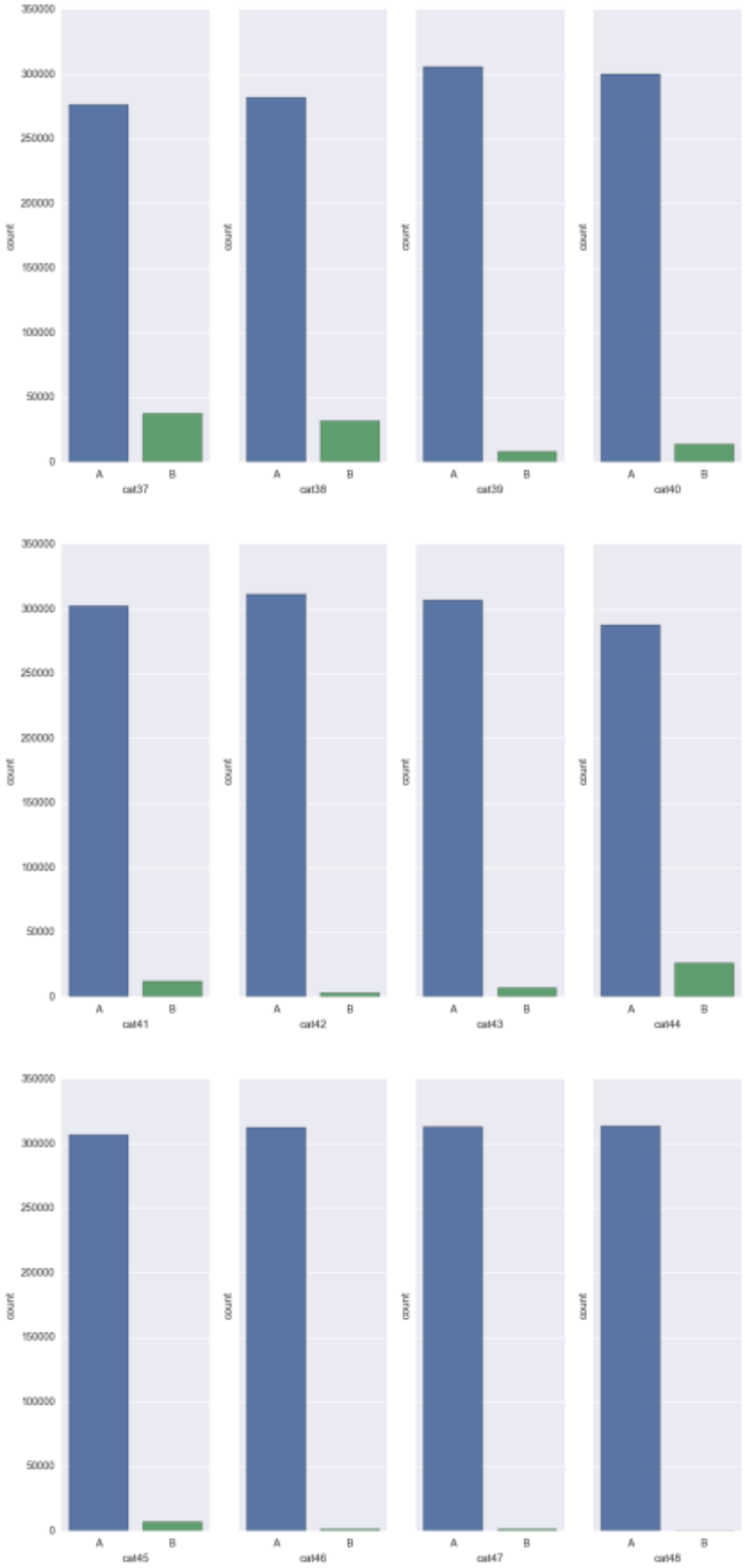
```
data = train_test[categorical_features]
cols = data.columns

# Count plot for all categorical features in a 29x4 grid
n_cols = 4
n_rows = 29
for i in range(n_rows):
    fg,ax = plt.subplots(nrows=1,ncols=n_cols,sharey=True,figsize=(12,
8))
    for j in range(n_cols):
        sns.countplot(x=cols[i*n_cols+j], data=data, ax=ax[j])
```

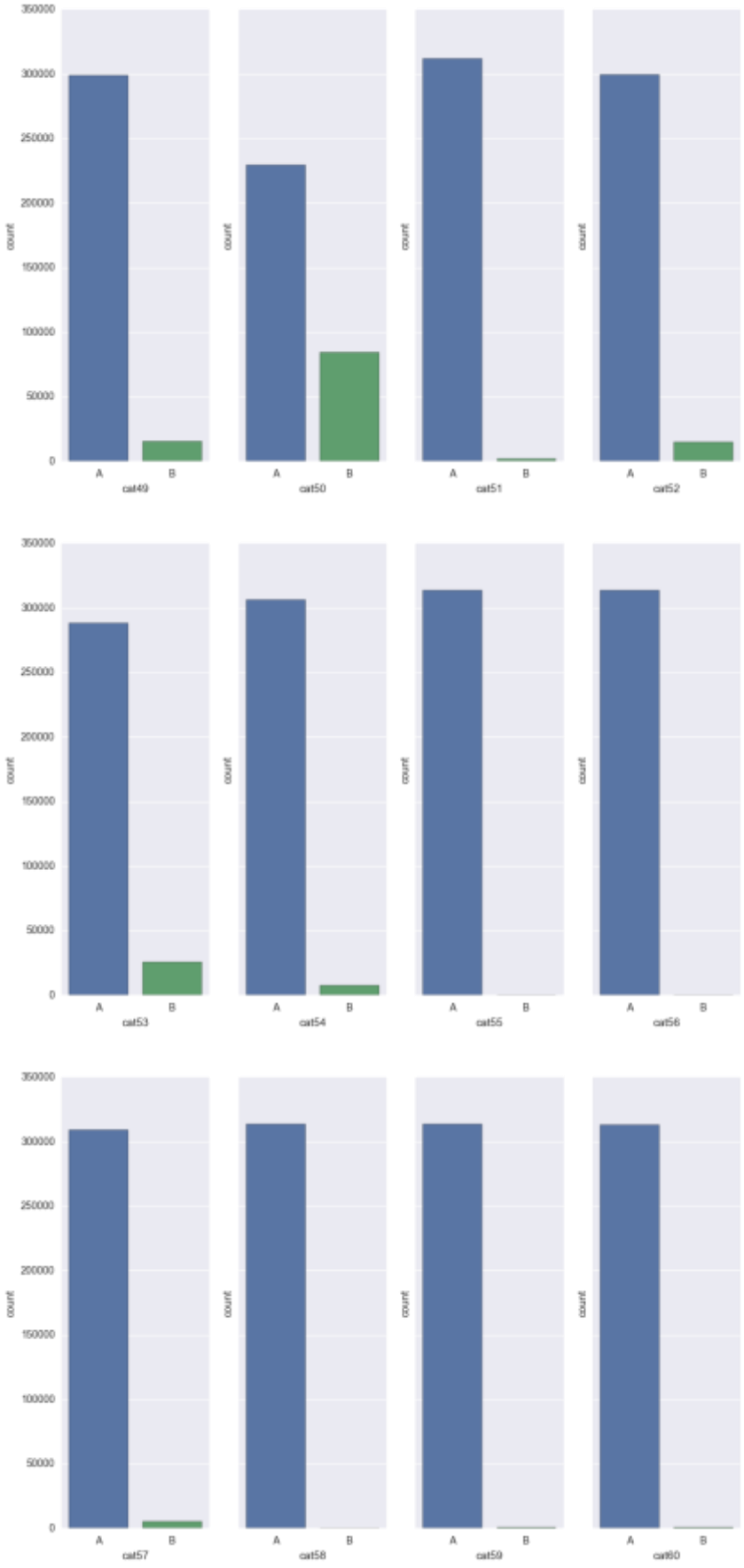


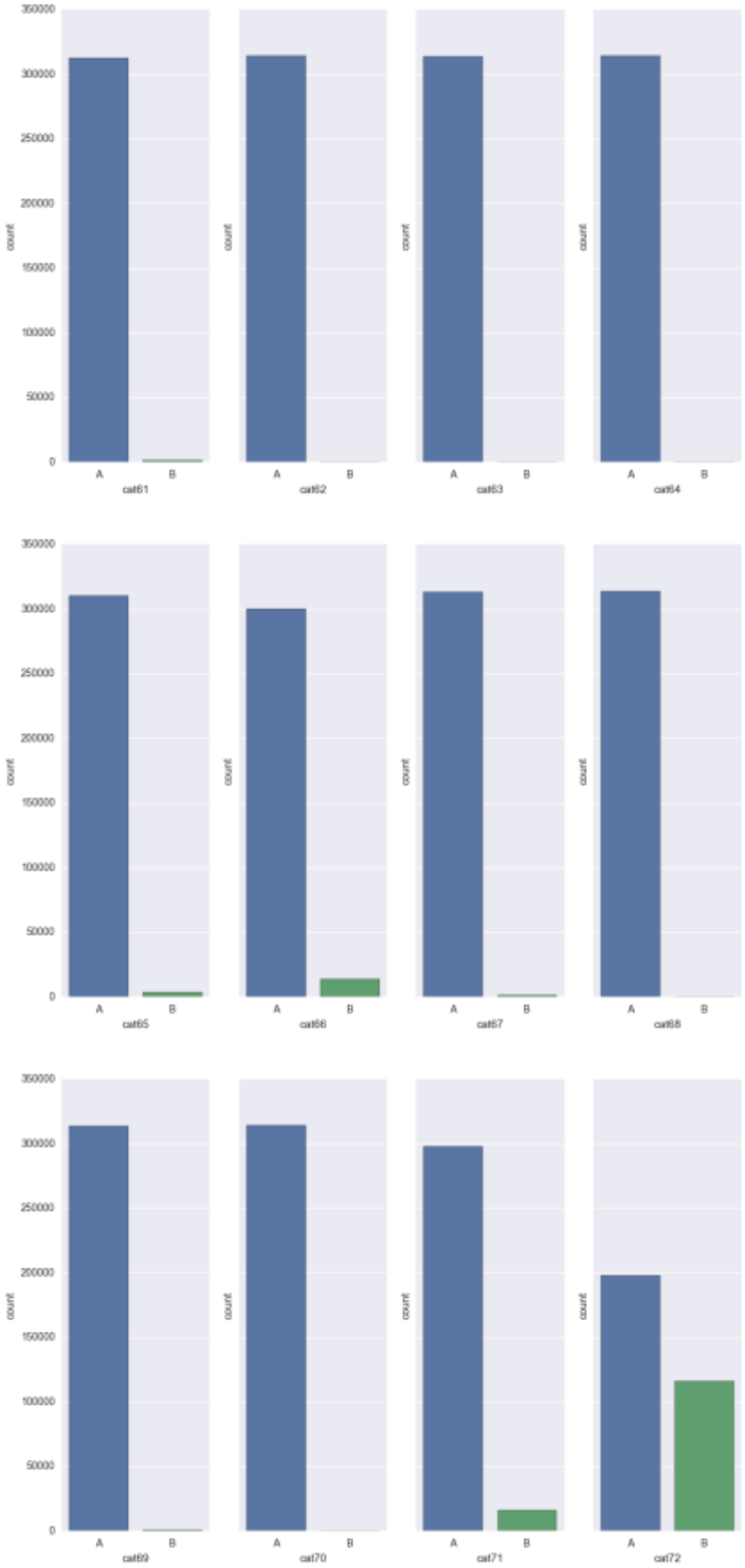

count
cat1
cat2
cat3
cat4
cat5
cat6
cat7
cat8
cat9
cat10
cat11
cat12

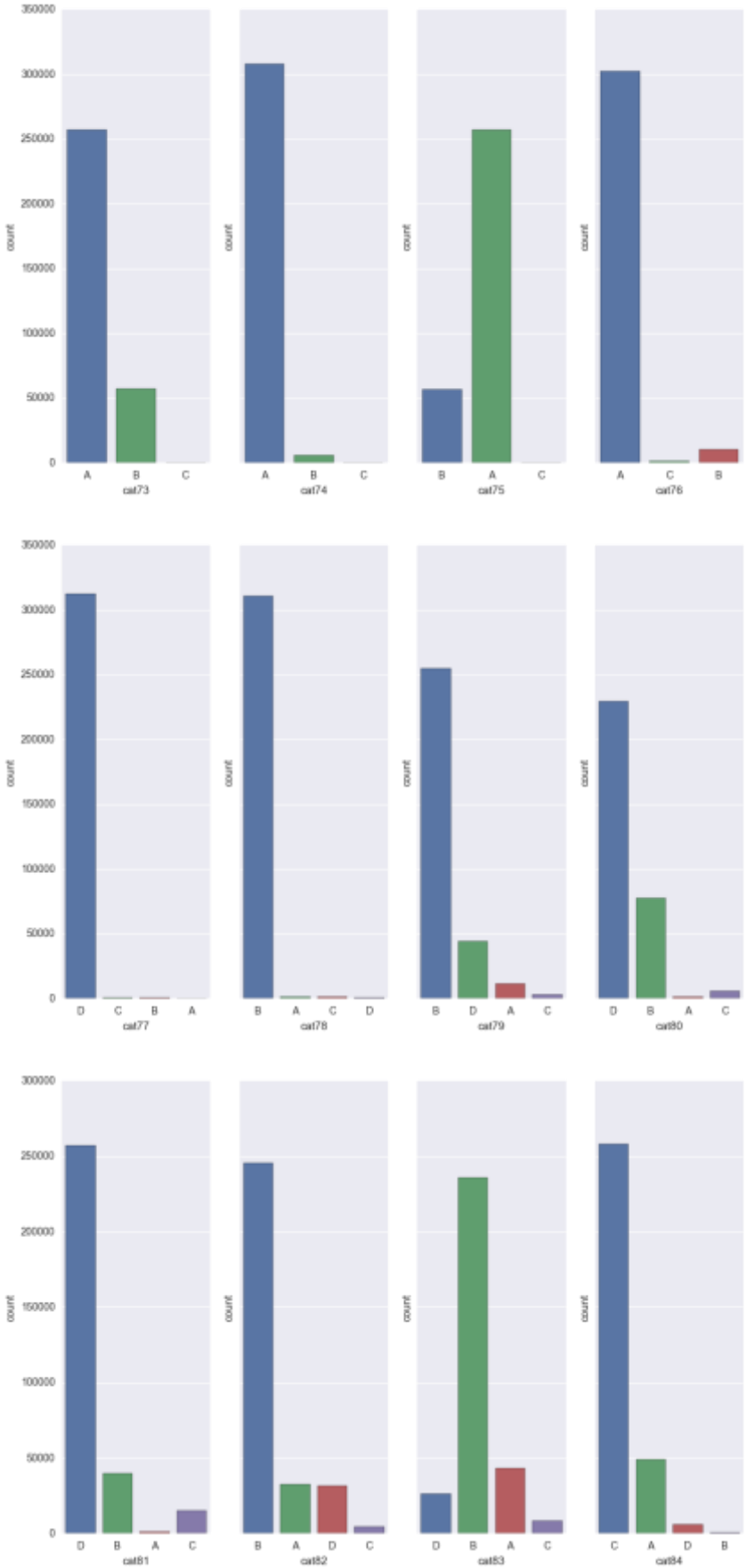

count
cat13
cat14
cat15
cat16
cat17
cat18
cat19
cat20
cat21
cat22
cat23
cat24
A
B

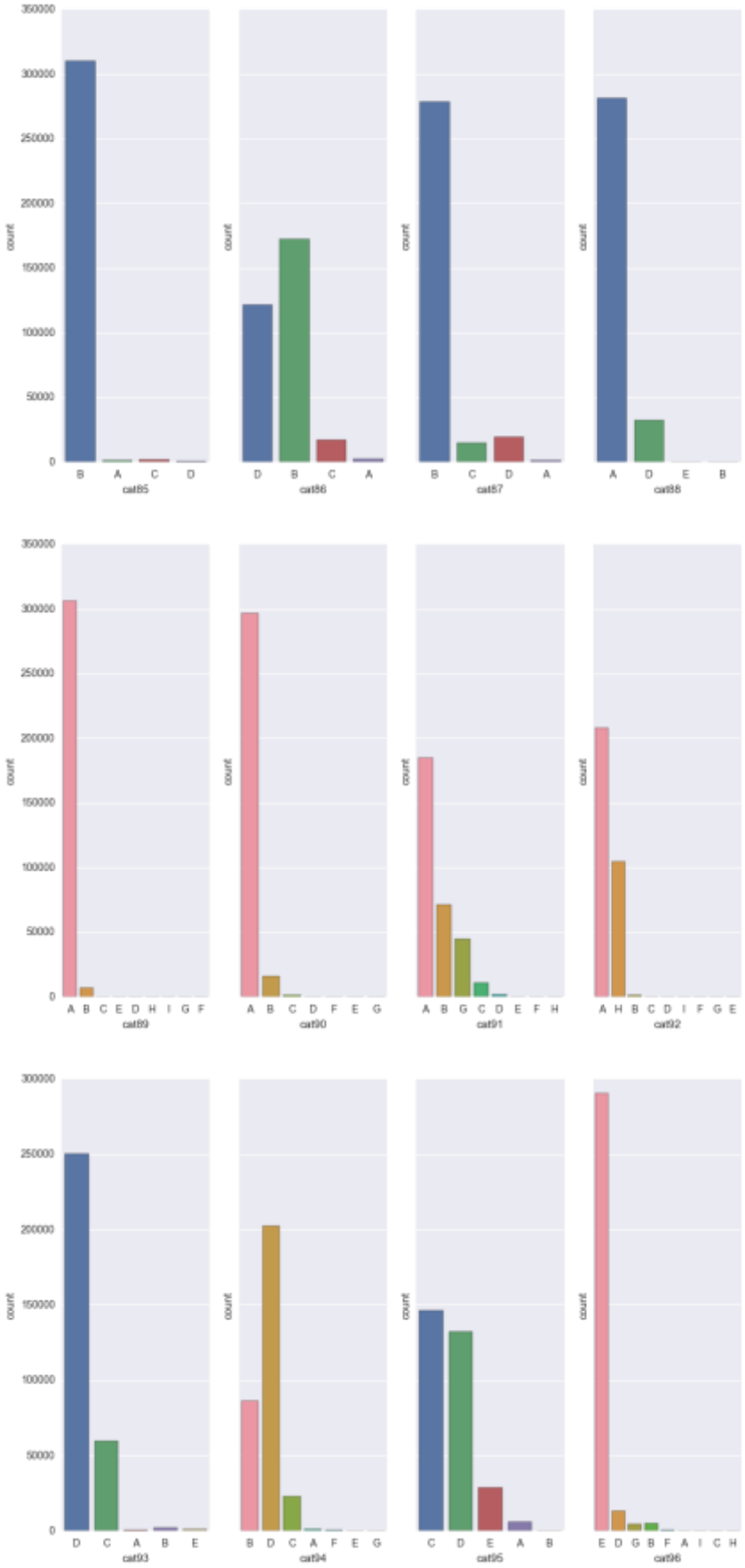

count
A
B
cat25
cat26
cat27
cat28
cat29
cat30
cat31
cat32
cat33
cat34
cat35
cat36

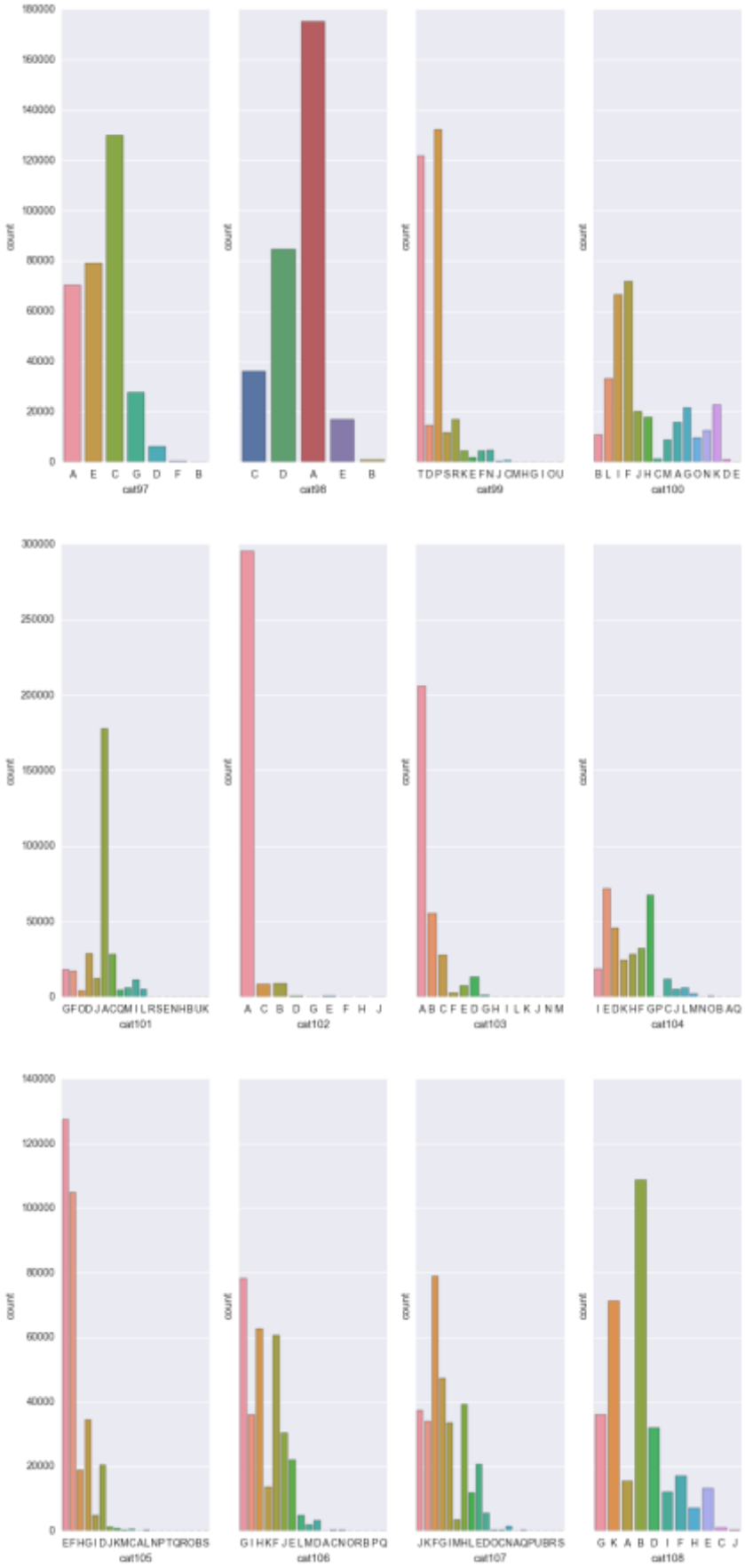

count
A
B
cat37
cat38
cat39
cat40
cat41
cat42
cat43
cat44
cat45
cat46
cat47
cat48

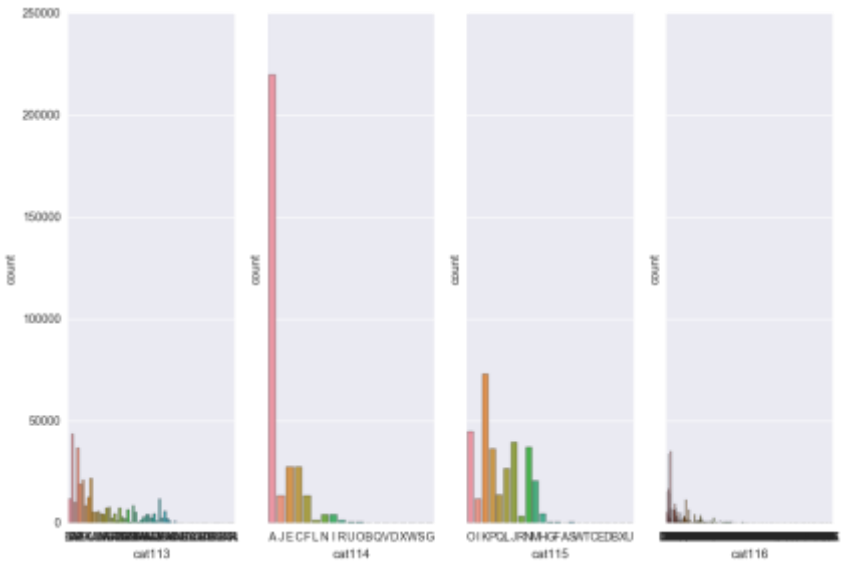

count
A
B
cat49
cat50
cat51
cat52
cat53
cat54
cat55
cat56
cat57
cat58
cat59
cat60


350000
300000
250000
200000
150000
100000
50000
0
count
A
B
cat61
cat62
cat63
cat64
cat65
cat66
cat67
cat68
cat69
cat70
cat71
cat72


count
cat73
cat74
cat75
cat76
cat77
cat78
cat79
cat80
cat81
cat82
cat83
cat84

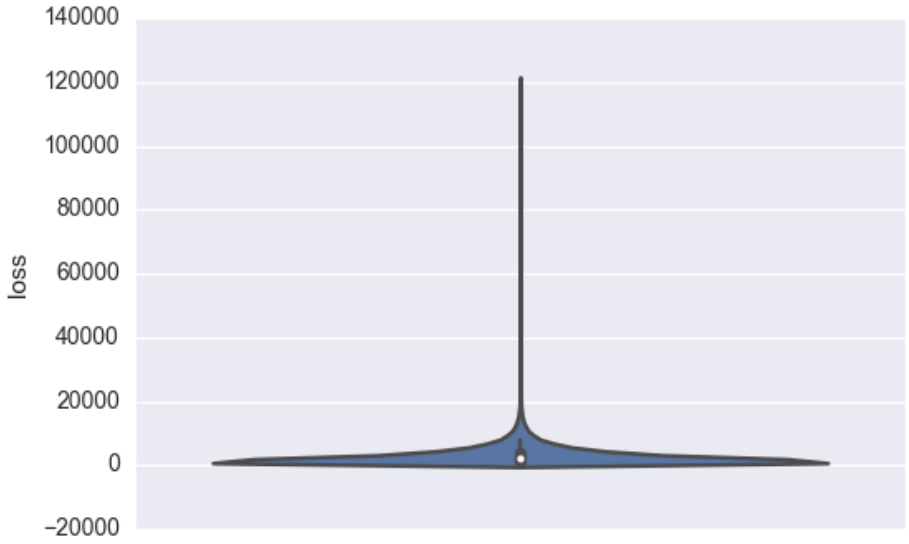

count
cat85
cat86
cat87
cat88
cat89
cat90
cat91
cat92
cat93
cat94
cat95
cat96


count
A E C G D F B
cat97
C D A E B
cat98
T D P S R K E F N J O M H G I O U
cat99
B L I F J H C M A G O N K D E
cat100
G F O D J A C Q M I L R S E N H B U K
cat101
A C B D G E F H J
cat102
A B C F E D G H I L K J N M
cat103
I E D K H F G P C J L M N O B A Q
cat104
E F H G I D J K M C A L N P T Q R O B S
cat105
G I H K F J E L M D A C N O R B P Q
cat106
J K F G I M H L E D O C N A Q P U B R S
cat107
G K A B D I F H E C J
cat108



2.2.3. Loss Target Variable

```
sns.violinplot(data=train,y="loss")
plt.show()
```



2.2.4. Conclusions

Numerical Features:

- cont1 has many values close to 0.5
- cont2 has a pattern where there are several spikes at specific points
- cont5 has many values near 0.3
- cont14 has a distinct pattern. 0.22 and 0.82 have a lot of concentration

Categorical Features:

- cat1 to cat72 have only two labels A and B. In most of the cases, B has very few entries
- cat73 to cat 108 have more than two labels
- cat109 to cat116 have many labels

Loss Target Variable:

- loss distribution must be preprocessed

2.3. Algorithms and Techniques

As the numerical features are skewed, we will apply a boxcox transformation to reduce the skewness. It's necessary to transform the categorical features in numbers to be able to use it in the learning models. We will standardize the data as well, including the target variable.

For academic purpose, two different regression models will be tested and one will be chosen to be tuned. First, we will apply the data to a generalized linear model called Bayesian Ridge Regression and then use an ensemble method called Gradient Tree Boosting.

2.4. Benchmark

The Kaggle competitions presents some results as benchmarks and we will use it as well. Those values are measured as Mean Absolute Square:

- All Zeros Benchmark: 3,019.71
- Random Forest Benchmark: 1,217.52

After predicting the values of the target variable 'loss' with the test data set, we must send to Kaggle the results and receive the score, as we don't have access to the true values.

3. METHODOLOGY

3.1. Data Preprocessing

3.1.1. Transform Data

Numerical Features

As we can see in section 3.1.5, all columns, including 'loss' target variable area skewed, except cont 3. This is a particularly important analysis for preprocessing the data for linear models. To correct the skewness, the boxcox transformation will be applied.

Categorical Features

The categorical features values are characters. To better apply this features to the model, we will factorize this values to numbers.

Loss Target Variable

As it is skewed, we will apply the boxcox transformation, as we did with numerical features.

```
from scipy.stats import boxcox

def aply_boxcox(df, cols, fact):
    skewed_feats = df[cols].apply(lambda x: skew(x))
    skewed_feats = skewed_feats[abs(skewed_feats) > fact].index
    for feat in skewed_feats:
        df[feat], lam = boxcox(df[feat]+1)

def factorize_features(df, cols):
    for col in cols:
        df[col] = pd.factorize(df[col], sort=True)[0]

start = time()

aply_boxcox(train_test, numeric_features, 0.25)
loss, loss_lam = boxcox(loss)
factorize_features(train_test, categorical_features)
train_rows = train_test.loc[features.index]
test_rows = train_test.loc[test.index]

end = time()
print "Transform data in {:.1f} seconds.".format(end - start)
```

```
Transform data in 19.7 seconds.
```

3.1.2. Standardize Data

After applying transformation, we need to standardize features by removing the mean and scaling to unit variance

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the transform method.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance, many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

```
from sklearn.preprocessing import StandardScaler
```

```
def scale_data(X, scaler=None):
    if not scaler:
        scaler = StandardScaler()
        scaler.fit(X)
    X = scaler.transform(X)
    return X, scaler

start = time()

_, scaler = scale_data(train_test)
train, _ = scale_data(train_rows, scaler)
test, _ = scale_data(test_rows, scaler)
y, scaler = scale_data(loss.reshape(-1, 1))

end = time()
print "Scale data in {:.1f} seconds.".format(end - start)
```

```
Scale data in 1.7 seconds.
```

3.2. Implementation

As the data was transformed and scaled, we will use R2 Scorer to evaluate and tune the model. Then, we will predict the values of the test data set and submit to the Kaggle competition that will inform us the error measured in Mean Absolute Error.

First, we will define a function that will be used for both models and then will be used to tune the best parameters for the chosen model.

This function defines a scorer, split the data into 80% train and 20% test sets, apply a grid search with the parameters informed for the specific model and then return the best estimator. We will use 6 processors at the same time (n_jobs=6).

3.2.1. Define Grid Search Function

```
from sklearn.metrics import make_scorer
from sklearn.metrics import r2_score
from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV

def fit_model(X, y, estim, params):
    score = make_scorer(r2_score, greater_is_better=True)
    sss = ShuffleSplit(len(X), n_iter = 10, test_size = 0.20,
random_state = 86401)
    grid = GridSearchCV(estimator=estim, param_grid=params,
scoring=score, cv=sss, n_jobs=6)
    grid = grid.fit(X, y)
    return grid
```

First, we will fit the Bayesian Ridge regressor limiting the maximum number of interactions in 2, 4 and 8.

3.2.2. Fit Bayesian Ridge

```
from sklearn.linear_model import BayesianRidge

start = time()

bay = BayesianRidge()
param_bay = {'n_iter':[2, 4, 8]}
bay_grid = fit_model(train, y, bay, param_bay)

end = time()
print "BAY grid search in {:.1f} seconds.".format(end - start)
print "BAY best score:  {:.4f}".format(bay_grid.best_score_)
print bay_grid.best_estimator_
```

```
BAY grid search in 59.2 seconds.
BAY best score:  0.4766
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False, copy
_X=True,
       fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06, n_iter=2,
       normalize=False, tol=0.001, verbose=False)
```

In almost 1 minute, the algorithm returned that the best number of interactions is 2.

Now, we will fit the Gradient Boosting Regressor with the parameters:

- Number of estimators: 25
- Maximum depth: 8

We will not use multiple parameters for the grid search as this algorithm takes a very long time to run.

3.2.3. Fit Gradient Boosting Regressor

```
start = time()

gbr = ensemble.GradientBoostingRegressor()
param gbr = {'n estimators' : [25], 'max depth' : [8], 'random state' :
[864]}
gbr_grid = fit_model(train, y, gbr, param_gbr)

end = time()
print "GBR grid search in {:.1f} seconds.".format(end - start)
print "GBR best score:  {:.4f}".format(gbr_grid.best_score_)
print gbr_grid.best_estimator_
```

```
GBR grid search in 3685.6 seconds.
GBR best score:  0.5078
GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1,
loss='ls',
             max_depth=8, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=25,
presort='auto',
             random_state=864, subsample=1.0, verbose=0,
warm_start=False)
```

After 1 hour, we have a 0.5078 score. As it's bigger than Bayesian Ridge score, we will use this algorithm to tune our model.

3.3. Refinement

To tune the model, we will use different values for the maximum depth parameter: 6, 8 and 10.

```
start = time()

gbr = ensemble.GradientBoostingRegressor()
param_gbr = {'n_estimators' : [25], 'max_depth' : [8, 6, 10],
'random_state' : [864]}
gbr_grid = fit_model(train, y, gbr, param_gbr)

end = time()
print "GBR grid search in {:.1f} seconds.".format(end - start)
print "GBR best score:  {:.4f}".format(gbr_grid.best_score_)
print gbr_grid.best_estimator_
```

```
GBR grid search in 3157.9 seconds.
GBR best score:  0.5188
GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1,
loss='ls',
             max_depth=10, max_features=None, max_leaf_nodes=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=25,
presort='auto',
             random_state=864, subsample=1.0, verbose=0,
warm_start=False)
```

With a maximum depth of 10, the result was very good and the score improved from 0.5078 0.5188, more than 0.01.

Now we will try a different set of parameters for maximum depth and number of estimators:

- N_estimators: 50
- Max_depth: 9, 10 and 12.

```
start = time()

gbr = ensemble.GradientBoostingRegressor()
param_gbr = {'n_estimators' : [50], 'max_depth' : [9, 10, 12],
'random_state' : [864]}
gbr_grid = fit_model(train, y, gbr, param_gbr)

end = time()
print "GBR grid search in {:.1f} seconds.".format(end - start)
print "GBR best score:  {:.4f}".format(gbr_grid.best_score_)
print gbr_grid.best_estimator_
```

```
GBR grid search in 11586.8 seconds.
GBR best score:  0.5424
GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1,
loss='ls',
             max_depth=10, max_features=None, max_leaf_nodes=None,
```

```
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=50,
presort='auto',
            random_state=864, subsample=1.0, verbose=0,
warm_start=False)
```

Again, our score improved to 0.5424 and the best value for max-depth is 10.

Now we will increase the number of estimators to 200, as we know that in ensemble methods, more is better.

```
start = time()

gbr = ensemble.GradientBoostingRegressor()
param_gbr = {'n_estimators' : [200], 'max_depth' : [10], 'random_state' :
[864]}
gbr_grid = fit_model(train, y, gbr, param_gbr)

end = time()
print "GBR grid search in {:.1f} seconds.".format(end - start)
print "GBR best score:  {:.4f}".format(gbr_grid.best_score_)
print gbr_grid.best_estimator_
```

```
GBR grid search in 11756.7 seconds.
GBR best score:  0.5457
GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.1,
loss='ls',
            max_depth=10, max_features=None, max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=200,
            presort='auto', random_state=864, subsample=1.0, verbose=0,
            warm_start=False)
```

The final score is 0.5457.

4. RESULTS

4.1. Model Evaluation and Validation

After training the model and defining the best parameters, we use the estimator that was trained to predict the 'loss' values for each row in the test set. The values predicted will be transformed and scaled, so we have to undo all the transformations, as next:

4.1.1. Predict Values

```
start = time()

y_gbr = gbr_grid.best_estimator_.predict(test)
y_gbr = scaler.inverse_transform(y_gbr)
y_gbr = np.exp(np.log(loss_lam*y_gbr+1)/loss_lam)

end = time()
print "GBR predict in {:.1f} seconds.".format(end - start)
```

```
GBR predict in 0.6 seconds.
```

To send the data predicted to the Kaggle contest, we have to save the predicted data in a CSV with a specific format.

4.1.2. Save Predicted Data

```
start = time()

df = pd.DataFrame()
df.insert(0, 'id', list(test_raw.index))
df.insert(1, 'loss', list(y_gbr))
df.to_csv('Z:/data/allstate/submission.csv',index=False)

end = time()
print "GBR predict in {:.1f} seconds.".format(end - start)
```

4.2. Justification

After submitting the predicted result to Kaggle, we have a score of 1,146.35.

As we have no domain knowledge at all of the problem, we can't say for sure if the solution is significant enough to have adequately solved the problem. But we have some clues:

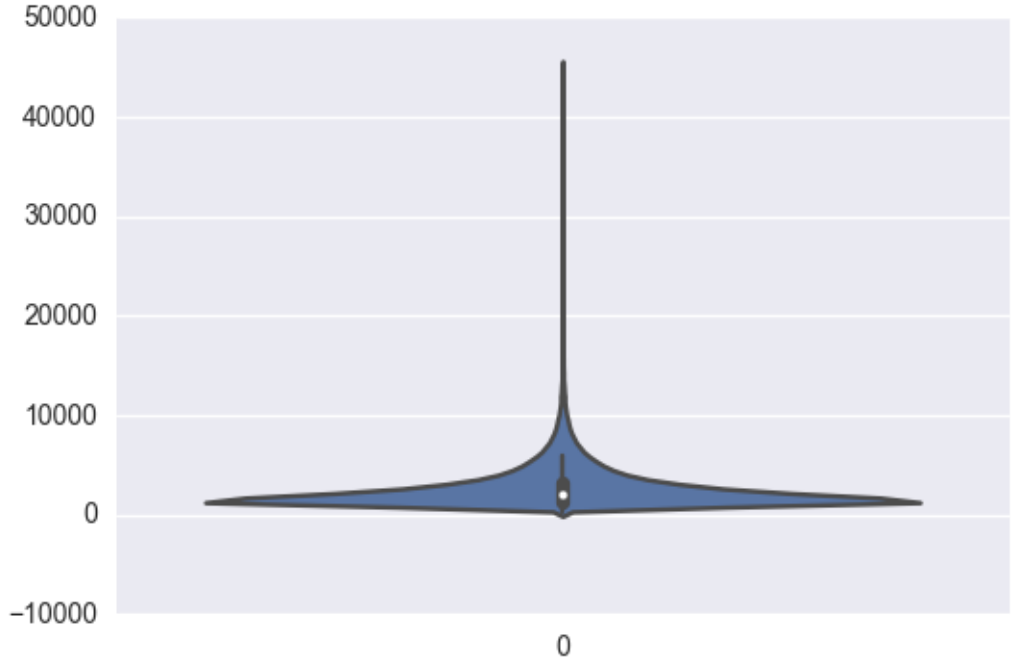
- Our score is better than the lowest Kaggle Benchmark score (1,217.52). So it's a good indicator that our model works well.
- But, considering that our Mean Absolute Error (1,146.35) is almost 38% of the mean value of the target variable 'loss' (3,037.34), we cannot say for sure it's a good solution for the problem.

5. CONCLUSION

5.1. Free-form Visualization

```
sns.violinplot(data=y_gbr)
plt.show()
print pd.DataFrame({'predicted': y_gbr}).describe()
```

```
          predicted
count  125546.000000
mean     2646.944917
std      2019.722474
min       248.732359
25%      1368.071047
50%      1994.295228
75%      3188.932779
max     45219.140168
```



As we can compare with the loss plot, the form of the y predicted is the same as the 'loss' target variable, as expected.

Y	'loss'	predicted
Mean	3037.337686	2646.944917
std	2904.086186	2019.722474
Min	0.670000	248.732359
25%	1204.460000	1368.071047
50%	2115.570000	1994.295228
75%	3864.045000	3188.932779
max	121012.250000	45219.140168

The parameters above are different from the 'loss' target variable:

- Mean: Lower
- Std: Lower
- Min: Higher
- 25%: Higher
- 50%: Lower

- 75%: Lower
- Max: Lower

We can infer from the table that this change is a reflection of the limited maximum depth of the Gradient Boosting Regressor to avoid overfitting.

5.2. Reflection

After analyzing the features and the data set, we found necessary to preprocess the numerical features because of its skewness and to preprocess the categorical features as well because it wasn't in a number format.

We have to preprocess the data all together (the train and data sets) so that we already prepare the test data for the prediction and then we just have to post process the predictions values (as we have had preprocess the target variable as well).

The target variable 'loss' is very concentrated in the first quartil of the data, but the numerical features are not. It may be because the features were given already preprocessed in a way that we don't know and/or because the categorical features have more influence in the 'loss' value than the numerical features.

The statistical linear model runs very fast, but the ensemble tree method doesn't.

5.3. Improvement

Because of the long time to fit the models to the data, it's very difficult for academic purpose to rewrite some of the code in time to get a better result. But next we can list some improvements:

- Outliers removal: We can remove the outliers and measure the improvement of the result.
- Dummy variables: We can transform the categorical features into dummy variables and delete from the train_test data the columns that only exists in the test data set. As it will not exist in the train data set, the model will have no information about it at all.
- Features reduction: We can apply a feature reduction algorithm as a PCA to avoid overfitting, but we can avoid it by tuning the model and limiting its parameters.