Capstone 2
Carsten Bruckner
Jan 2022

Genotyping SNP Classification

Contents

Problem statement

Evaluate whether a model can be created that predicts a binary quality category for each SNP's genotype calls better than existing feature 'OriginalCT', which is an available categorization of observations into various recommended and non-recommended performance bins.

The quality category to predict is "quality_binary_good", where observations have "1" category if [Call Rate (CR) > 95%] AND [Concordance to 1000Genomes reference data (CC) > 98.5%]. Otherwise quality_binary_good = "0".

Criteria for success

This exercise will be considered successful if the model has a higher accuray predicting quality_binary_good than prediction accuracy using only one of the engineered features, namely 'OriginalCT.recommended_True'.

	available quality_binary_good (is probeset CR>=95% and CC>=98.5%)	
predicted quality_binary_good	0	1
0	good	bad
1	bad	good

	OriginalCT.recommended_True	
predicted quality_binary_good	0	1
0	good	bad
1	bad	good

Background

Individual single nucleotide polymorphisms (SNPs) in a genome usually report one of three possible answers:

- 2 copies of the reference allele
- 1 copy of reference allele, 1 copy of variant allele
- 2 copies of variant allele.

("Alleles" are usually represented by A,C,G,T nucleotides.)

An example of a SNP that has frequent changes in the human population is rs688:

Various technologies are available to measure genomic DNA. Microarray technology uses

1. polymerase chain reaction to amplifiy fragments of DNA,
2. let each single-stranded fragment bind to a known complementary sequence bound in a known location on a glass surface
3. add a fluorescent dye to the bound fragment
4. measure the total fluorescent signal with a microscope

Microarray technology doesn't count individual molecules, but instead measures a separate signal for each allele. Sometimes it is difficult to measure a SNP accurately, because the specific sequence bound to the glass microarray doesn't always bind specifically to ONLY the one target sequence we're looking for. After all, the human DNA sequence is cut into hundreds of thousands of small pieces, individually amplified, and then those billions++ molecules are floating in solution over the sequence of interest.

There's a day-long "fishing" exercise where many of those molecules bump into the sequence on the glass surface, and only "stick" if the matching sequence is close enough. Some sequences in the human DNA are similar enough that side-reactions happen at a high enough rate to start interfering with the specific sequence we're trying to capture and measure.

Additionally microarray surface defects, sample DNA quality, and variations in test condition can all introduce issues making it more difficult to get quality test results.

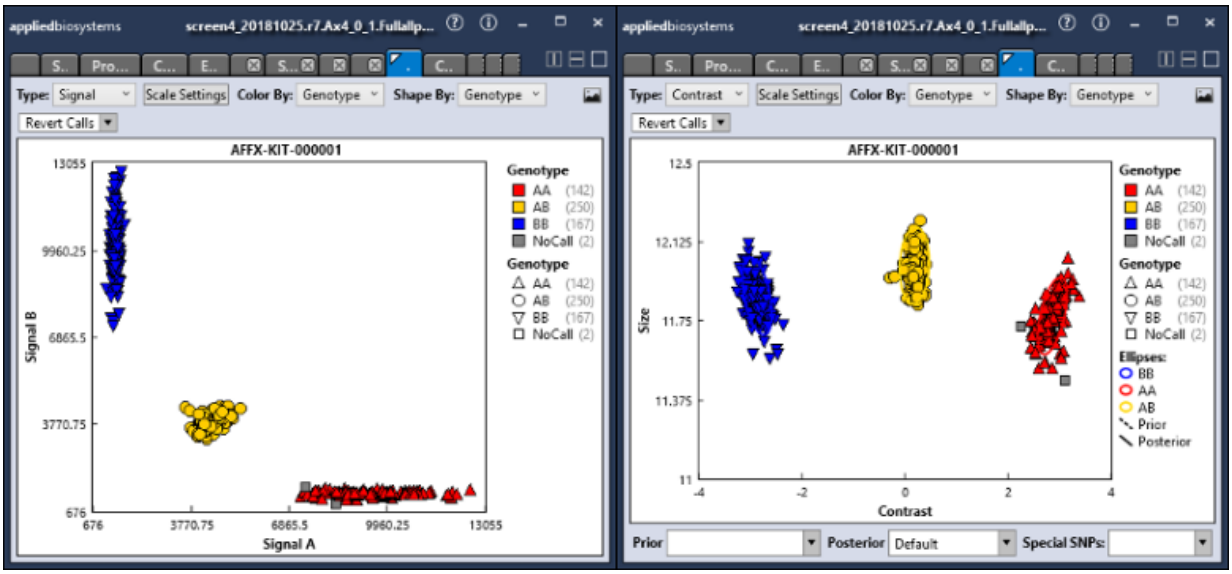
Therefore it is important to do quality control on the results to decide which measurements are reliable enough to be used later in an analysis pipeline.

When multiple DNA samples are tested at once, it is possible to use all the data across samples to better understand when an individual measurement is suspect. This is because for most SNPs, we expect the signal data to segregate in only 1 to 3 classes across all samples:

- a group of samples that report 2 copies of allele 1 (let's say "A/A", same nucleotide from each parent)
- a group of samples that report 1 copy of each allele (let's say "A/C", different nucleotides from each parent)
- a group of samples that report 2 copies of allele2 (let's say "C/C", same nucleotides from each parent)

(There are exceptions for X,Y chromosomes, cases where there are sequence duplications/deletions, and where a 3rd and even 4th allele are possible, but I'm generalizing.)

An example of signal data for one SNP across many samples, labeled by predicted call:

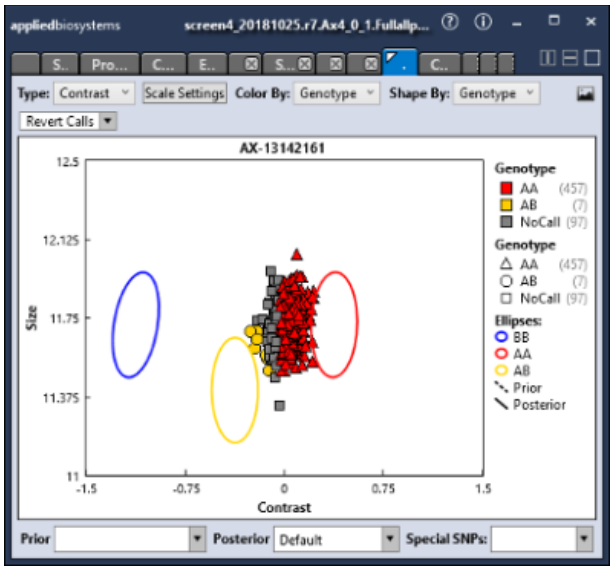


Labeling and summary statistics is more easily doing on transformed data, where

X axis = contrast = log2(signalA/signalB)

Y axis = size = 0.5*log2(signalA*signalB)

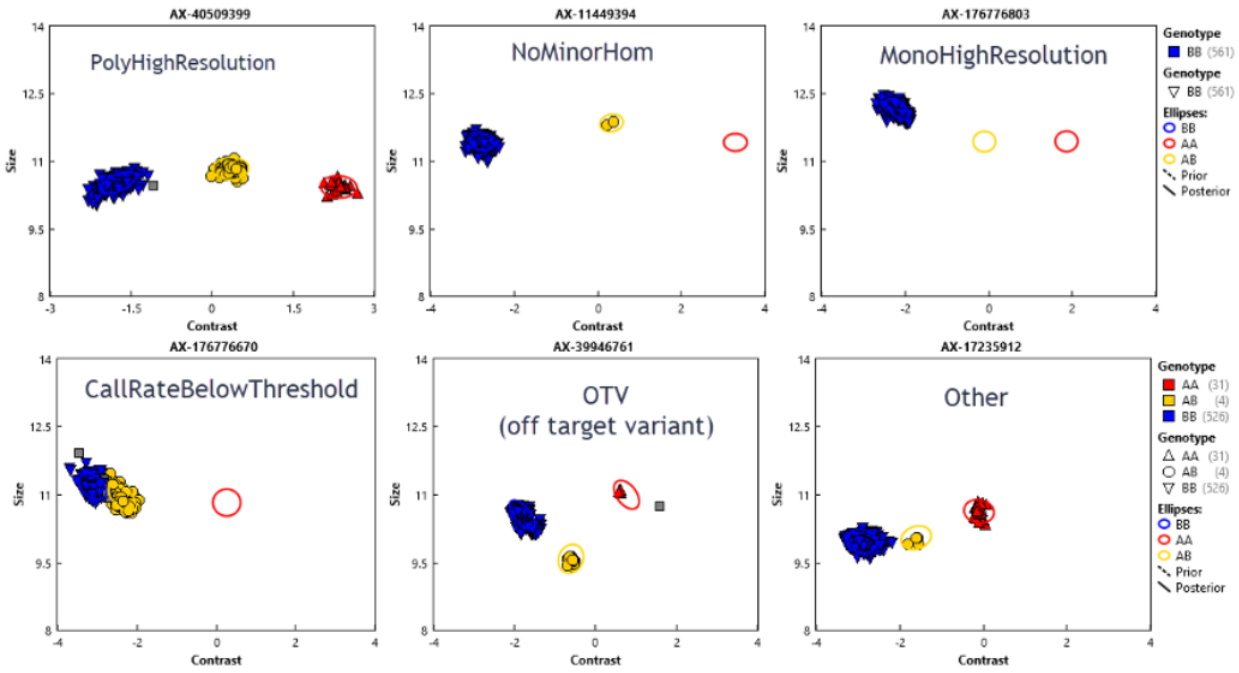
Labeling can sometimes go wrong, leading to mis-genotyped data. While it is impossible to guarantee accurate calls, it is possible to identify SNPs that might have miscalls in some samples in the analysis batch. If the miscall rate is high enough, ANY results from that SNP are suspect and should not be used downstream. An example of a suspect SNP:



One way to identify errors in genotyping is to test the DNA using multiple technologies concurrently and flag disagreements. If three tests are done on the same sample, majority voting can select the "expected" call, which can then identify the measurement that is "wrong", because it is in the minority. This procedure is expensive and is not done routinely. Instead, it is usually done when validating a technology, or when following up on a significant result.

The expected genotype calls, or "ground truth", is available for samples that are widely available and have been tested by multiple technologies. Crucially, for THIS CAPSTONE PROJECT, a genotyping data set is available that has expected genotype calls for a significant fraction of measured SNPs. Therefore, it is possible to segregate the measurements of SNPs into those that are poor quality, and those that are good quality.

In standard practice, most batches of analyzed samples won't have available ground truth. Therefore it is important to identify metrics that can predict whether a SNP's calls across a batch of samples are reliable or not. Because of this need, the proposed data set already includes a feature that classifies each SNP into one of several "good" and "bad" categories. This classification step requires a complicated decision tree where various other metrics (features) are compared to predefined thresholds. Example classifications are shown for some SNP examples:



The existing classification feature 'OriginalCT' is meant to have a high correlation between whether [a member of a "recommended" ConversionType] and [a SNP with accurate genotypes].

Recommended ConversionTypes are: PolyHighResolution, NoMinorHom, MonoHighResolution

Bad ConversionTypes are: CallRateBelowThreshold, OTV, Other

Data

Summary statistics from >800000 SNP probesets were generated from non-public Thermo Fisher Scientific microarray data run on three 96-well plates of samples from coriell.org: HAPMAPPT01, HAPMAPPT02, HAPMAPPT03. The actual input data for data cleaning are standard summary statistics for each SNP generated by ps-metrics.exe and ps-classification.exe engines, along with "concordance to 1000 Genomes" performance statistics. The data was generated using the Million Veterans Program biobanking version 1.0 array https://www.thermofisher.com/blog/biobanking/the-million-veteran-program-biobanking-to-the-max/. Example data quality control analysis of data from this array: https://www.cell.com/ajhg/pdf/S0002-9297(20)30080-X.pdf.

A partially-obfuscated identification of the source data is from project PRO100409_MVPEF409_SAX, with raw inputs generated on July 14, 2021 originally managed here:

".\Falcon_usr\amittal\MVP\MVPEF\Output_allps\genotype-inliers\filtered\Ps.performance.txt"

".\Falcon_usr\amittal\MVP\MVPEF\Output_allps\CC_ignoreNC_probeset_CC.txt"

Both source files were then zipped into:

https://github.com/cabruck/DataScienceCapstoneTwo/blob/main/raw_data/Output_allps.zip

The input “Ps.performance.txt” file contains standard metrics routinely generated by automated SNP quality control. Each row is summary statistics for one SNP (“probeset_id” is the unique identifier). rows can be grouped into different categories of markers:

- standard diploid probesets that measure exactly 2 alleles. This is large majority of observations in dataset.
- multiallelic probesets: measure more than 2 alleles.
- special chromosome probesets: statistics computed only only a subset of samples (only female for X, only male for Y),
- or only measure mitochondrial DNA, which has different probeset properties
- other probesets that can report more than standard 3 genotype calls (haploid and zero copy number calls)

Many of the measured features of the probesets are specific to these special categories of markers, which means that there is high missingness of data in many features that are only computed for some categories of markers. Ther,efore this file needs to be cleaned up.

The input “CC_ignoreNC_probeset_CC.txt” file contains observed accuracy metrics for a subset of probesets for which accuracy can be computed. The surrogate measurements of accuracy are various concordance metrics, of which the primary metric of interest is “CC” (overall concordance or agreement. for the probeset_id vs an independent technology prediction of the genotype calls). This file has some rows where CC is missing, and has other rows where the concordance calculation is based on limited data (“n” samples < 50, out of over 250 possible samples measured). Therefore this input file must also be cleaned.

The “CC” metric is one of the two key metric we’re trying to predict with a machine learning model (see earlier Problem Statement section for how quality_binary_good is calculated).

Method

In the interest of time, most of the data cleaning and exploratory data analysis was performed with the TIBCO Spotfire Analysis desktop application, version 11.6.

Data splitting into test-training sets, replacement of empty values, one-hot encoding, and model building and evaluation were performed using Jupiter Lab 3.2.1, with packages scikit-learn 1.0.2 and pandas 1.3.4.

Data cleaning and feature engineering

Full data cleaning details are available here:

https://github.com/cabruck/DataScienceCapstoneTwo/blob/main/reports/DataWranglingNotes.pdf

In short, the data cleaning steps amount to

- remove 10% of original observations (probeset rows) because they belong to at least one of these categories:
    - more than 2 alleles
    - on non-autosomal chromosomes (X,Y,mitochondrial)
    - non-diploid calls
    - no values for essential Concordance component of the predictor variable
    - unreliable values for essential Concordance component of the predictor variable
- remove features only computed for the non-standard observations that are being removed
- remove features derived from remaining features, so not linearly independent
- remove sparse features highly correlated with less-sparse features, like sparse HomFLD vs FLD.
- remove annotation features with high cardinality (like "affy_snp_id")

Feature engineering steps (mostly done during exploratory data analysis) amount to:

- reset to Null in new columns any values that exist only due to Bayesian prior values and not due to the data being measured (for example, if there are no samples for AA cluster, then AA.meanX and AA.meanY metrics are not real values). New columns were created with .clean appended to name, original columns were removed.
- replace highly related count metrics n_AA, n_AB, n_BB with het_frac = n_AB/(n_AA + n_AB + n_BB).
- create quality_binary, where observations have "good" category if [Call Rate (CR) > 95%] AND [Concordance to 1000Genomes reference data (CC) > 98.5%].

Empty value replacement

Final feature engineering was done as part of the RandomForest notebook, linked in a later section. The original idea for delaying this step was that some models can handle sparse data, so I only wanted to fill in empty values if absolutely needed. The empty values are not missing, but instead cannot be computed. I believe it is inappropriate to replace these empty values with just the mean or median of remaining values. So I used domain knowledge to replace empty values with what I think are appropriate values. Data filling was done using the following custom function, applied immediately building Random Forest models.

```
def mydata_fillna(X):
    X_filled = X.copy()

    fill_lo_columns = ['BB.meanX.clean','HomRO']
    fill_0_columns = ['AB.meanX.abs_clean','AA.varX.clean','AB.varX.clean','BB.varX.clean', \
                      'AA.varY.clean','AB.varY.clean','BB.varY.clean']
    fill_hi_columns = ['AA.meanX.clean','FLD','HetSO','MMD']

    X_filled[fill_lo_columns] = X[fill_lo_columns].fillna(-999)
    X_filled[fill_0_columns ] = X[fill_0_columns ].fillna( 0  )
```

```
    X_filled[fill_hi_columns] = X[fill_hi_columns].fillna( 999)

    X_filled['AA.meanY.clean'] =
X['AA.meanY.clean'].fillna(X[['AB.meanY.clean','BB.meanY.clean']].min(axis=1))

    X_filled['BB.meanY.clean'] =
X['BB.meanY.clean'].fillna(X[['AA.meanY.clean','AB.meanY.clean']].min(axis=1))

    X_filled['AB.meanY.clean'] =
X['AB.meanY.clean'].fillna(X[['AA.meanY.clean','BB.meanY.clean']].max(axis=1))


    return X_filled
```

Exploratory data analysis

Full EDA is available here:

https://github.com/cabruck/DataScienceCapstoneTwo/blob/main/reports/ExploratoryDataAnalysis.pdf

EDA was used to select which new features to engineer, and which to remove. EDA was used to hypothesize which features might be most useful and which might be removed if doing manual selection of features for model building.

One large pairplot of the remaining continuous features, colored by the benchmark OriginalCT.Recommended metric, is linked below. A number of features look informative for predicting quality_binary, since quality_binary is highly correlated with OriginalCT.Recommended.

https://github.com/cabruck/DataScienceCapstoneTwo/blob/main/reports/EDA_pairplot2.png

Features used for model building:

feature	type
CR	float
FLD	float
HetSO	float
MMD	float
het_frac	float
MinorAlleleFrequency	float
H.W.p-Value	float
AA.meanX.clean	float
AB.meanX.abs_clean	float
BB.meanX.clean	float
HomRO	float
AA.meanY.clean	float
AB.meanY.clean	float
BB.meanY.clean	float
meanY	float
Hom.meanY.delta	float

AA.varX.clean	float
AB.varX.clean	float
BB.varX.clean	float
AA.varY.clean	float
AB.varY.clean	float
BB.varY.clean	float
OriginalCT.recommended_True	boolean
OriginalCT_AAvarianceX	boolean
OriginalCT_AAvarianceY	boolean
OriginalCT_ABvarianceX	boolean
OriginalCT_ABvarianceY	boolean
OriginalCT_BBvarianceX	boolean
OriginalCT_BBvarianceY	boolean
OriginalCT_CallRateBelowThreshold	boolean
OriginalCT_MonoHighResolution	boolean
OriginalCT_NoMinorHom	boolean
OriginalCT_OTV	boolean
OriginalCT_Other	boolean
OriginalCT_PolyHighResolution	boolean
Nclus_1	boolean
Nclus_2	boolean
Nclus_3	boolean

Algorithms and machine learning

The assignment called for evaluating three different algorithms for binary classification. I had time to evaluate only one: Random Forest. Random Forest uses consensus classification from a collection of decision trees, where each tree is built with some randomness. The randomness is which subset of features are considered at each decision node, when selecting the single most useful feature at that node.

The good and bad classes are highly imbalanced, with only about 4.3% of observations in a "bad" category (see table below). Additionally, there are different kinds of observations (OriginalCT), with some types being relatively rare.

probeset count:	quality_binary	
OriginalCT	good	bad
NoMinorHom	498455	5747
PolyHighResolution	228775	6801
MonoHighResolution	27866	952
Other	8175	16756

ABvarianceY	2322	551
ABvarianceX	2251	256
AAvarianceY	1383	75
BBvarianceX	1359	67
BBvarianceY	1300	142
OTV	927	1027
AAvarianceX	812	74
CallRateBelowThreshold	NaN	2082
total	773625	34530
	95.7%	4.3%

When splitting the data into test and training sets, a 70:30 train:test split was done for each stratified OriginalCT and quality bin group (12x2 groups), in order to retain proportions of each group in both train and test sets.

https://github.com/cabruck/DataScienceCapstoneTwo/blob/main/reports/train_test_split_part3.html

It is easier for RandomForest to accurately call the minor 'bad' class if the number of 'good' and 'bad' classes in the training set are more balanced. Since there are a large number of observations in the training set (n = 541536 "good", n = 24172 "bad"), I decided to subsample the "good" class to only 1 of every 15, but retain all of the "bad" class. After resampling, the subsampled training set has n = 36102 "good", n = 24172 "bad", for a more balanced 60:40 split.

Parameter Tuning

A number of hyperparameters were evaluated. After non-thorough testing of various values for the following parameters, defaults were used for these:

- max_leaf_nodes: default None worked best
- min_samples_split: default 2 worked best
- min_samples_leaf: default 1 worked best
- criterion: default 'gini' pretty similary to 'entropy'

Every combination (3x4) of the following parameters was evaluated:

- max_depth = [15, 25, None] # None is default
- n_estimators = [25, 50, 100, 800] # 100 trees is default

Five-fold cross-validation of each set of evaluated parameters was performed, which means five random forests were trainined on a different 80% subset of the training set, and evaluated on the remaining 20% of the training set, in order to get model average qualtiy score for each evaluated parameter set.

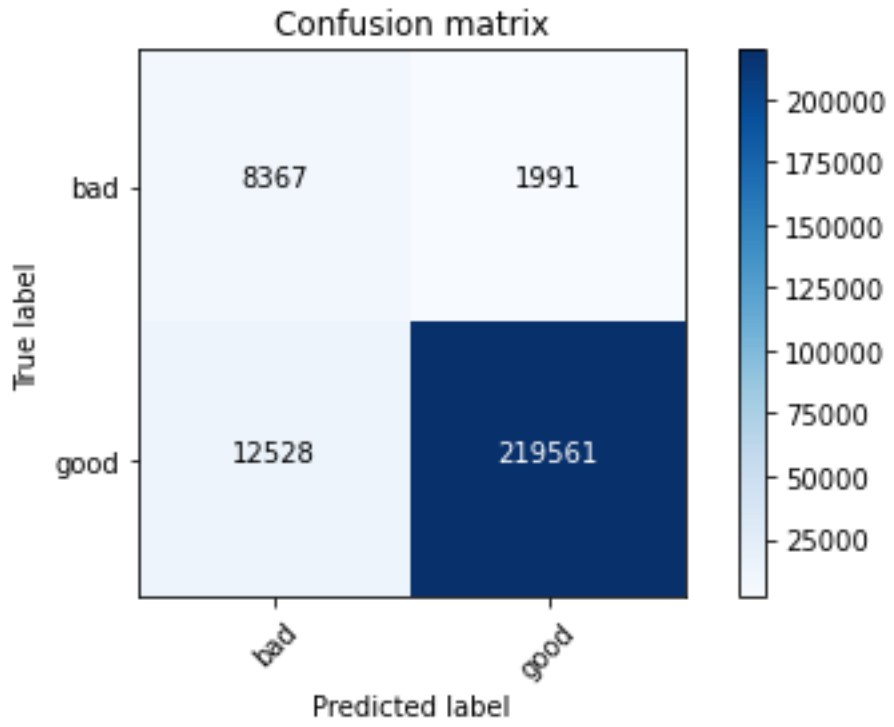
The scoring metric for ranking all evaluated models is 'balanced_accuracy', which maximizes the average of each class's accuracy. Since the classes were already mostly balanced by subsampling, the hyperparameter search may have picked the same parameter set if I had instead used the default 'accuracy' scoring metric.

The balanced_accuracy across the full hyperparameter search did not change much, ranging from 87.4% to 87.8%. Using more trees than the default (800 vs 100) showed an improvement, as did allowing the tree to have a larger depth (25 vs 15). Within the evaluated parameter space, the best model used these parameters:

```
best estimator: RandomForestClassifier(max_depth=25, n_estimators=800, random
_state=1)
```

Predictions

The best model was evaluated on the independent test set (30% of observations withheld from training).

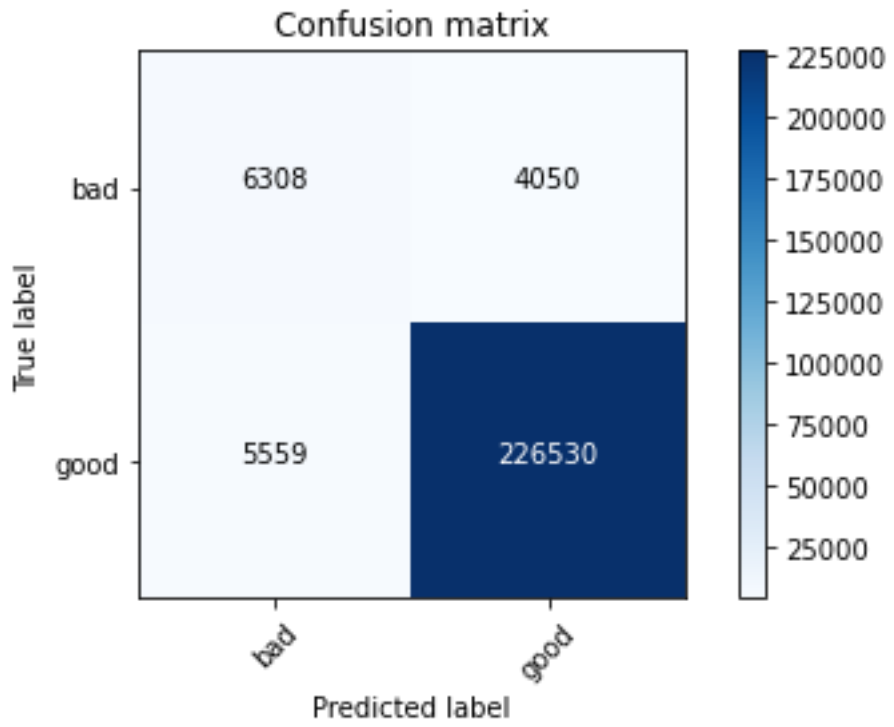


	precision	recall	F-beta	support (n)
bad	0.400431	0.807781	0.535437	10358.0
good	0.991013	0.946021	0.967995	232089.0

```
balanced_accuracy: 87.7 %
```

'bad' class recall is ~81% in test set.

As a benchmark, let's see if the model did a better job predicting the True label in the test set than just using the single feature 'OriginalCT.recommended_True' as the Predicted label:



	precision	recall	F-beta	support (n)
bad	0.531558	0.608998	0.567649	10358.0
good	0.982436	0.976048	0.979231	232089.0

balanced_accuracy: 79.3 %

The benchmark 'OriginalCT.recommended_True' feature only identified ~61% of the 'bad' observations (recall) in the test set.

In comparison, the final RandomForestClassifier(max_depth=25, n_estimators=800, random_state=1) identified ~81% of the bad observations (recall). This amounts to a 2-fold reduction in number of bad observations mislabeled as 'good'.

53% of the observations labeled as bad by benchmark 'OriginalCT.recommended_True' feature are actually bad. In comparison, 40% of the observations labeled as bad by the RandomForest model are actually bad.

So relative to the benchmark, the RandomForest classifier throws out more good observations (5.4% vs 2.6%) when trying to grab more bad observations (81% vs 61%).

Since the original criteria for success was to improve overall accuracy, not balanced accuracy, this RandomForest model could be considered a failure vs just using an existing feature. However, if it is more important to reject bad observations than it is to keep good observations, the RandomForest model can be considered an improvement.

Future Improvements

Directions I haven't yet explored:

- rejecting less informative input features from model building
- replacing empty values with different values
- evaluating different algorithms, like Logistic Regression, Gradient Boost, K nearest neigbors, and even basic Decision Tree for explainability