Fordham University

Department of Computer and Information Science

Predicting Hospital Readmission Rates Using Machine Learning Algorithms

CISC 6930: Data Mining

Instructor: Dr. Yijun Zhao

Submitted By:

Movses Musaelian

Revathi Bhuvaneswari

Youfei Zhang

Fall 2018

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Context of the Problem

The problem at hand deals with a real world medical dataset from the UCI machine learning lab. One issue that can be quite important in the medical field is the prior knowledge and prediction of whether patients will return after a visit in the hospital. This can have implications for how certain patients can be treated to ensure an optimal recovery phase. With the advent of easy to access, high level computing capabilities and the application of several data mining techniques, the medical field has become a critical opportunity point for data science, given the amount of data that is recorded and the implications that smarter analysis can have for patients and doctors alike.

The dataset we are working with includes instances of patient encounters at 130 US Hospitals and medical facilities over a period of 10 years. At a glance, the features of the data include demographic/biographical information about the patient, in addition to information about the patient's diagnosis, treatments and medications. The question that our model is to address is whether a person will be readmitted within 30 days after being discharged from the hospital.

Qualititatively, one can presume that certain features may drive the probability that a given patient returns. For example, elderly patients may very well be more likely to return or patients with certain diagnosis that require continuous attention. We of course also pay attention to the fact that every patient will likely have their own approach to how they deal with their medical care (e.g. certain patients being more careful than others and going back to hospital), and of course the natural supposition that each patient has a different body, different health status, and so forth. Before even starting our data exploration, it is important to thoroughly understand this context and asks ourselves such questions, which may later on affect how we apply our models. It is natural that the question of statistical independence of data points and collinearity of features will come up, and by assessing the context from a simply qualitative view guides us in those decisions later on in the process.

2. SOLUTION CONCEPT

2.1. Initial Approach and Thoughts

Having understood the context of the dataset and the problem at hand, we knew that cleaning the data would be a crucial task in our solution. There were several decisions we would have to make regarding the encoding of our features. Our goal was to synthesize the raw data in the most concise and efficient way possible, in order to avoid redundancies, and make sure that our features are robust enough. Given the fact that we had medical data at hand with mostly categorical features, we had several assumptions about what models could possible work and not work. We know from experience that Naive Bayes and Logistic Regression models are often used in such cases.

For example, with medical data, it's easy to see that one can have a prior knowledge, in this case the prior knowledge of one's medical or health history. This can serve as a powerful prior. For instance, one could assume that elderly patients with severe diagnoses will be much more likely readmitted than younger patients with less severe problems. In general, Naive Bayes theorem is often used in the medical field, where the question posed is not just the probability that patient X has disease Y, but the probability that patient X has disease Y given that patient X is a given age, has certain medical history, family history, and so forth.

At the onset, we knew that we would not use Linear Regression because of the nature of our data and the fact we were to predict a binary output, something Linear Regression is not optimal for. Logistic Regression, on the other hand, is geared for such cases. With this in mind, we would seek to further categorize our data, for example, utilizing data binning when necessary.

2.2. Possible Pitfalls

A few glaring pitfalls were visible to us from the onset even before delving deeper into the analysis. The first pitfall was the fact of imbalance of data, one of the biggest issues in this dataset. While we will talk more in detail about this imbalance and how we tried to remedy it, we recognized that the imbalance of such data occurs often in the medical world. For example, for cancer detection you may have in your data 99% benign, but 1% malignant. Obviously, one can get 99% accuracy by predicting all the instances as benign, but such a model would be very dangerous in the real world as it would miss all the malignant cases.

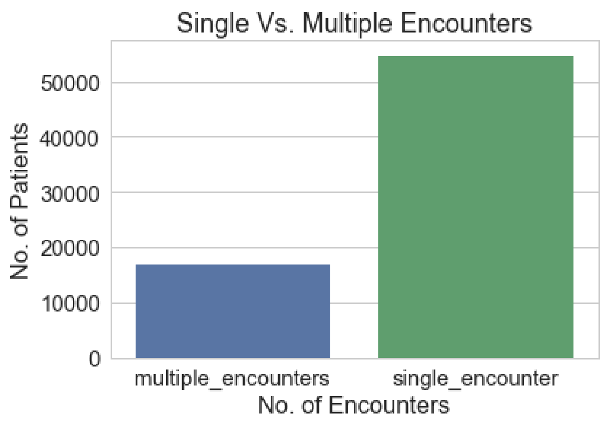
In our context, we don't have a severe danger of "misdiagnosis" since our prediction is simply related to a patient being readmitted or not. But that being said, we do not want to fall into that trap, because in the end of the day, it is our goal to have a robust and generalizable model, one that can be robust to unseen data and not be dependent on the balance of our predictive classes. Thus we knew, we would have to pay attention to the recall of our models and make sure that we had a healthy balance between accuracy and recall.

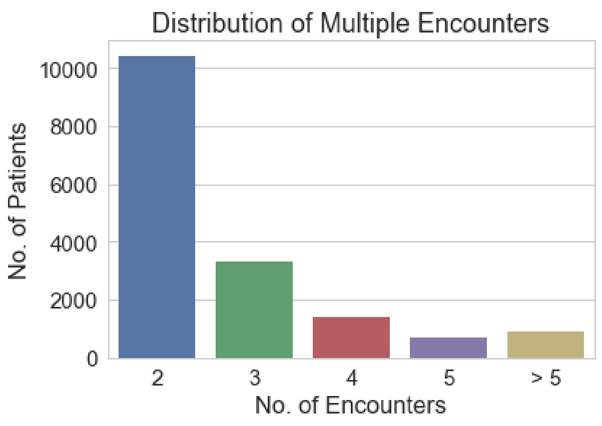
Another pitfall we recognized early on was the initial coding of the data. Real world data can of course be messy and we noticed how certain features had different formats and varying coding. Thus, it was imperative for us to firstly understand what all the features are telling us, and second, given exploratory data analysis, how crucial are they for our model.

3. EXPLORATORY DATA ANALYSIS

3.1. Dataset Description

The UCI dataset contains 101,766 unique hospital encounters (visits), where each encounter corresponds to a hospital admission record in which diabetes was coded as an existing health condition. The encounters correspond to 71,518 unique patients (id), implying that certain patients have had multiple encounters.





We thus easily see that most patients have made just one visit, with a considerable portion of two visits, and then a long tail. This fact gave us the reasoning that the data can be cleaned either by visit based or patient based. For example, do we want a unique patient dataset or simply leave it how it is, based on visits? This is a question we would address in our data cleaning later.

Additionally, the dataset also comprises of 49 features and 1 target variable (Readmitted), which provides detailed information on each patient encounter spread among certain categories.

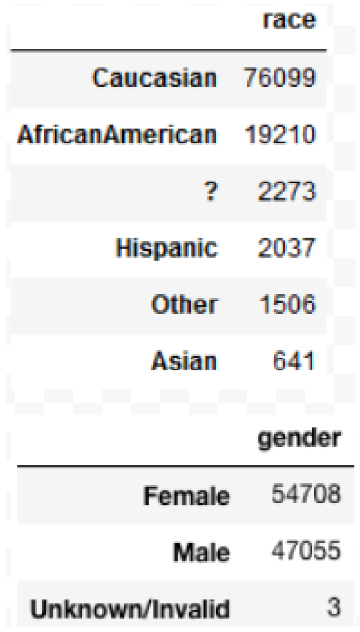
Feature Category	**Feature Type**	**Features Name (count)**
Unique IDs	*Numerical*	Encounter ID, Patient NBR (2)
Patient Demographics	*Categorical*	Race, Gender, Age (3)
Admission & Discharge Info	*Categorical*	Admission Type ID, Discharge Disposition ID, Admission Source ID, Medical Specialty, Payer Code (5)
Hospital Stay Metrics	*Numerical*	Time in Hospital, No. of Lab Procedures, No. of Procedures, No. of Outpatient Visits, No. of Emergency Visits, No. of Inpatient Visits (6)
Diagnosis Info	*Categorical*	Diagnosis-1, Diagnosis-2, Diagnosis-3 (3)
	Numerical	No. of Diagnoses (1)
Medical Metrics & Results	*Categorical*	Maximum Glucose Serum, A1C Results, Weight (3)
Medications Info	*Categorical*	23 Medications Features, Medication Change, Diabetes Medication (25)
	Numerical	No. of Medications (1)

3.2. Feature Analysis and Distributions

We decided to get a detailed look at each of the features and their distributions in order to be better prepared for the data cleaning task. We recognize that "?" / "Unknown" signifies missing data. For simplicity purposes, only the head rows are shown for few distributions below.

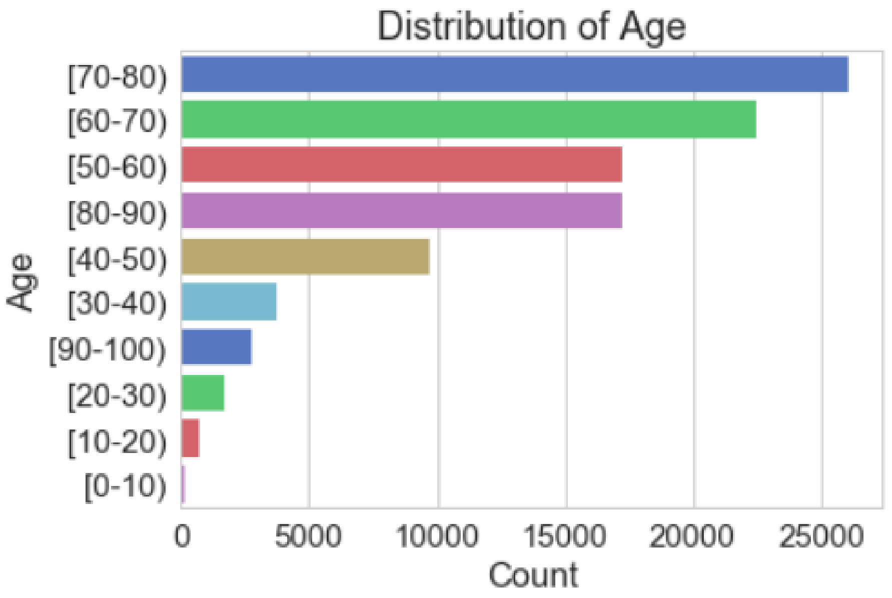
Patient Demographics Features - Race, Gender, Age

We notice that the demographic features are categorical strings. It can be seen that Race and Gender has missing values, and while Gender is fairly well-balanced, Race is more skewed towards 'Caucasian'. Additionally, for Age, there is a skew towards elderly patients, implying that we won't be needing so many bins for this feature, which will be addressed in our cleaning.



	race
Caucasian	76099
AfricanAmerican	19210
?	2273
Hispanic	2037
Other	1506
Asian	641

	gender
Female	54708
Male	47055
Unknown/Invalid	3



Admission & Discharge Information Features

The Admission Type, Discharge Disposition and Admission Source ID features are categorical coded columns, where each code corresponds to a specific sub-category. We recognize these may be important features in our model, but may optimize the coding of the variables by utilizing smarter binning and collapsing the values into fewer important codes.

	admission_type_id
1	53990
3	18869
2	18480

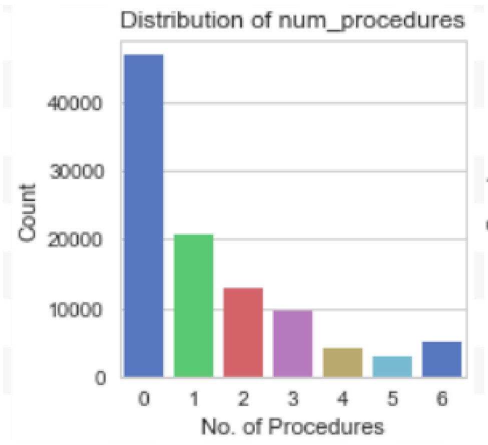
	discharge_disposition_id
1	60234
3	13954
6	12902

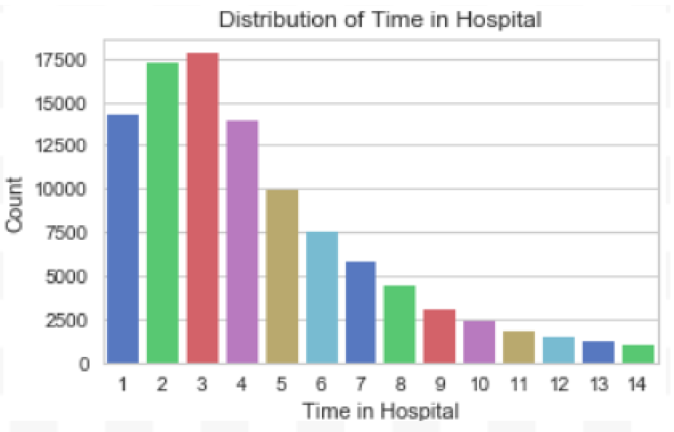
	admission_source_id
7	57494
1	29565
17	6781

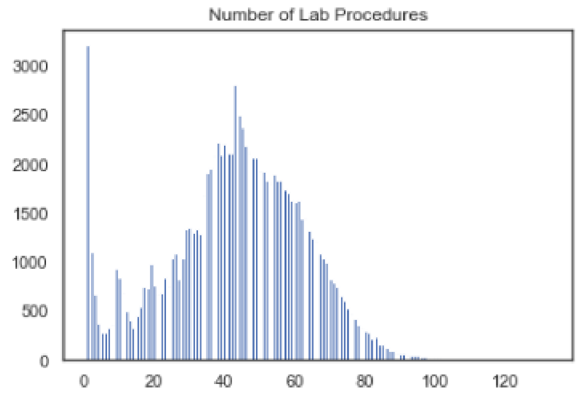
The Payer Code feature has 40,256 missing values. Given the high amount of missing values, and the fact that payer code doesn't seem relevant to our question, it becomes a candidate for removal. Similarly, Medical Speciality has 49,949 missing values, making it not useful for our modeling, suggesting that this could also be potentially removed from our dataset as well.

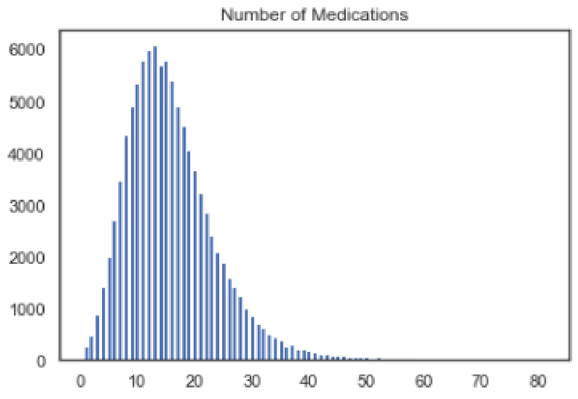
<u>Hospital Stay Metrics, No. of Medications and No. of Diagnoses</u>

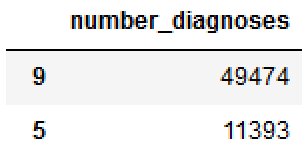
Six out of the eight numeric features fall under the Hospital Stay Metrics category, with the other two being No. of Medications and No. of Diagnoses. The distributions of the numeric features below gives us an insight into the importance of their role in determining hospital readmission rates as they signify the complexity of a patient's condition. For example, having higher value for these numeric features could suggest that the patient required additional care and extensive treatment methods, which in turn could lead to a higher readmission rate. It is however important to note that the distributions of certain features are concentrated towards a particular value, implying that they should be handled in a smarter way by balancing out the skewness.



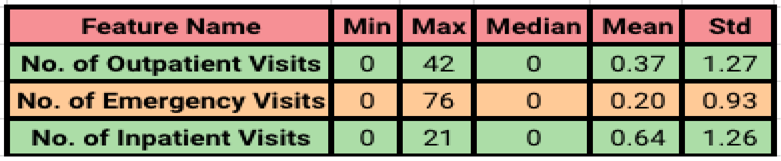








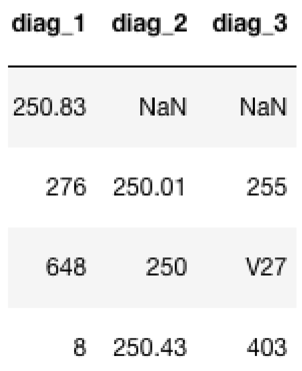
	number_diagnoses
9	49474
5	11393



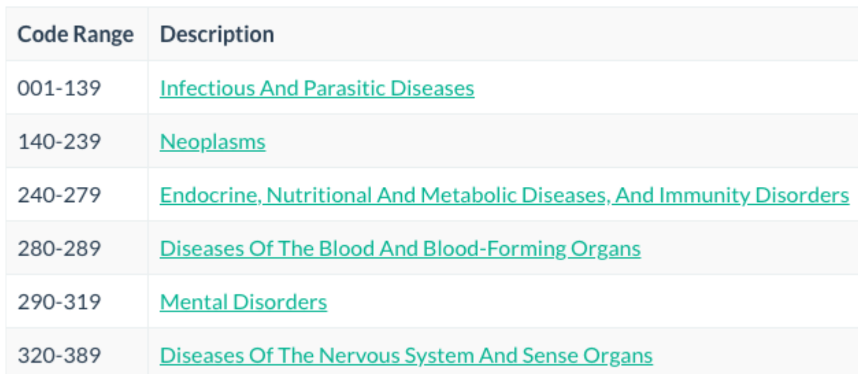
Feature Name	Min	Max	Median	Mean	Std
No. of Outpatient Visits	0	42	0	0.37	1.27
No. of Emergency Visits	0	76	0	0.20	0.93
No. of Inpatient Visits	0	21	0	0.64	1.26

Diagnoses Information

The features Diagnosis-1, Diagnosis-2 and Diagnosis-3 refer to primary, secondary and additional diagnoses respectively. Although at first glance they might look like numerical features, in reality, they are categorical nominal features, that correspond to the first three digits of ICD-9 codes. The distributions of the variables suggests that the three diagnoses have different values per encounter, and that they have missing values in them. Additionally, just Diagnosis-1 column has 717 distinct values, implying that it is important to collapse these into fewer sub-categories during the cleaning process by utilizing the categorization in ICD official website.



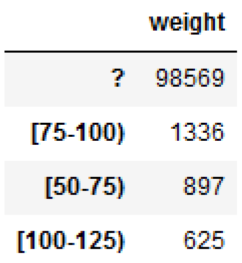
diag_1	diag_2	diag_3
250.83	NaN	NaN
276	250.01	255
648	250	V27
8	250.43	403



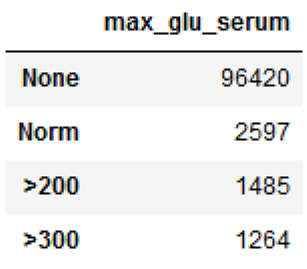
Code Range	Description
001-139	Infectious And Parasitic Diseases
140-239	Neoplasms
240-279	Endocrine, Nutritional And Metabolic Diseases, And Immunity Disorders
280-289	Diseases Of The Blood And Blood-Forming Organs
290-319	Mental Disorders
320-389	Diseases Of The Nervous System And Sense Organs

Medical Metrics & Results - Weight, Max Glucose Serum, A1C Result

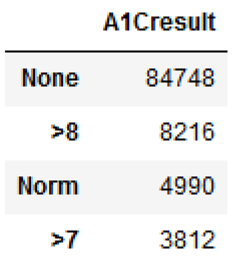
Weight is a categorically coded feature that has range of patient's recorded weight as values. It can be seen that the Weight feature has 98,569 missing values in it, which is more than 90% of the dataset. This makes it a candidate for removal during our cleaning process.



	weight
?	98569
[75-100)	1336
[50-75)	897
[100-125)	625



	max_glu_serum
None	96420
Norm	2597
>200	1485
>300	1264

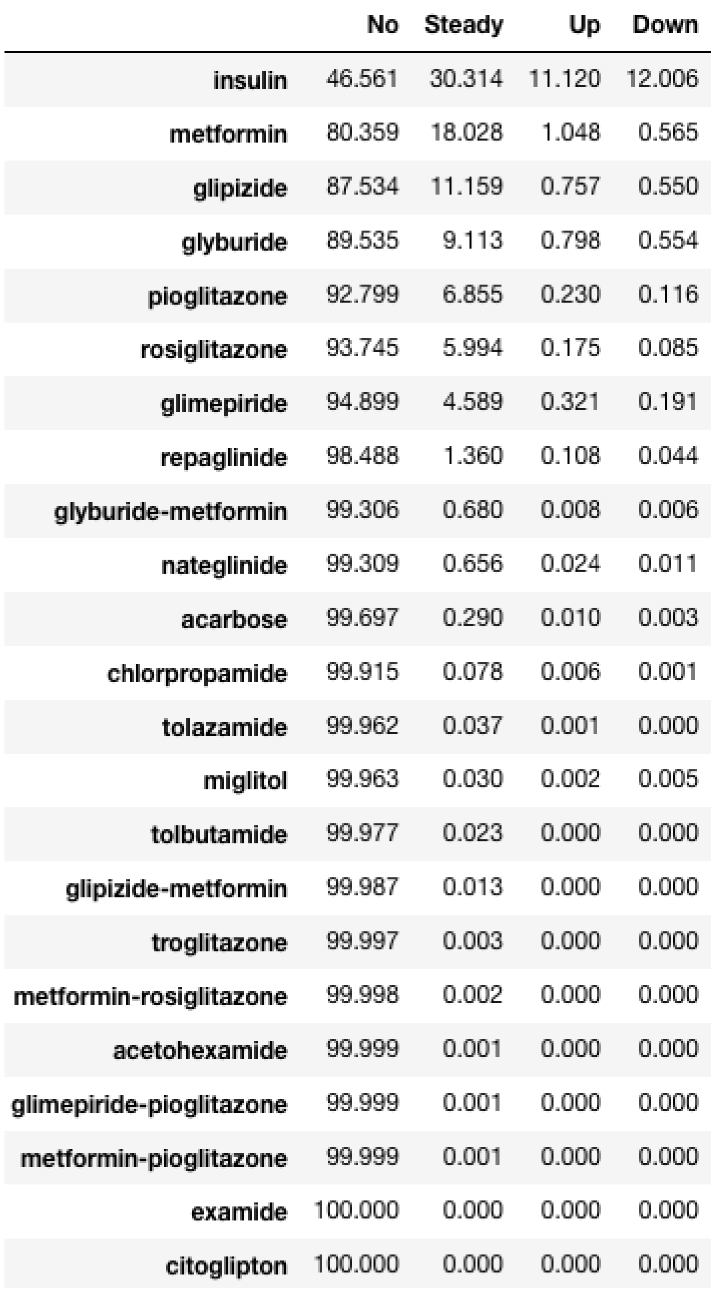


	A1Cresult
None	84748
>8	8216
Norm	4990
>7	3812

Max Glu Serum and A1C Result features are both categorical ordinal features that refers to the patient's results for both the tests. These two features could be critical for our modeling as a higher value for these results might indicate a more severe diabetes condition, thus increasing the probability of readmission for a particular patient. It is interesting to note that majority of the encounters have patients who have not taken the tests, which could also be because of our dataset containing information on patient's single / multiple encounters instead of unique patients.

<u>Medications Information - 23 Medications, Medications Change, Diabetes Medication</u>

We have 23 different diabetes medications features with similar categorical encodings. They are coded with ('Down', 'No', 'Steady', 'Up'), which is referring to the change in in dosage after the given visit. In the context of the problem, we suppose that if dosage of medicine has been increased, there is possibly greater chance of readmission, given that is indicative of a worsening condition. We can clearly see from the below distribution table that some medications give almost no information because of everything being in one class. Such medications are thus good candidates for removal as they add no needed information for the model.



	No	Steady	Up	Down
insulin	46.561	30.314	11.120	12.006
metformin	80.359	18.028	1.048	0.565
glipizide	87.534	11.159	0.757	0.550
glyburide	89.535	9.113	0.798	0.554
pioglitazone	92.799	6.855	0.230	0.116
rosiglitazone	93.745	5.994	0.175	0.085
glimepiride	94.899	4.589	0.321	0.191
repaglinide	98.488	1.360	0.108	0.044
glyburide-metformin	99.306	0.680	0.008	0.006
nateglinide	99.309	0.656	0.024	0.011
acarbose	99.697	0.290	0.010	0.003
chlorpropamide	99.915	0.078	0.006	0.001
tolazamide	99.962	0.037	0.001	0.000
miglitol	99.963	0.030	0.002	0.005
tolbutamide	99.977	0.023	0.000	0.000
glipizide-metformin	99.987	0.013	0.000	0.000
troglitazone	99.997	0.003	0.000	0.000
metformin-rosiglitazone	99.998	0.002	0.000	0.000
acetohexamide	99.999	0.001	0.000	0.000
glimepiride-pioglitazone	99.999	0.001	0.000	0.000
metformin-pioglitazone	99.999	0.001	0.000	0.000
examide	100.000	0.000	0.000	0.000
citoglipton	100.000	0.000	0.000	0.000

	change
No	54755
Ch	47011

	diabetesMed
Yes	78363
No	23403

The Change feature is a binary categorical variable that refers to whether there was change in the diabetic medication. It is in a way related to our previous medication features as a change in them should be reflected in this feature. This may be a useful feature for us. We see a fairly even split between No and Change. Similarly, the Diabetes Med feature is another binary feature indicating whether diabetes medicine was prescribed. Such a feature can be indeed indicative if a given patient will return, since it may signify a worsening of their condition if the medicine was prescribed. We can see that most of the encounters have diabetes meds prescribed.

<u>Target Variable - Readmitted</u>

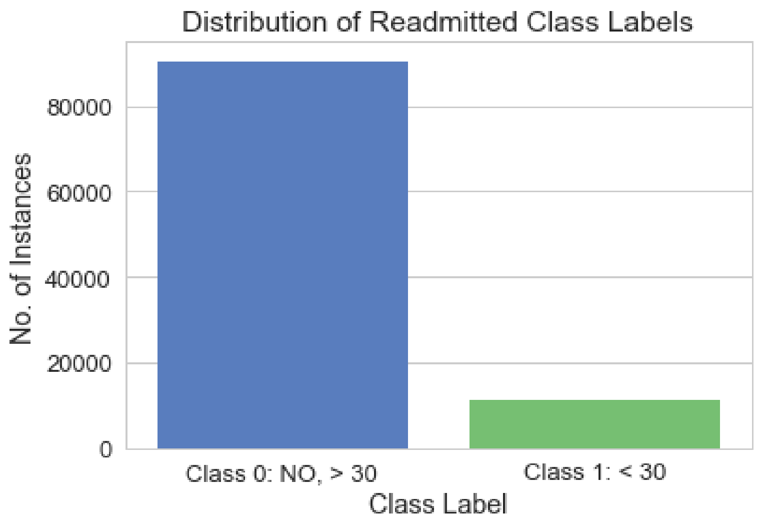
The Readmitted feature is the target variable whose class we want to predict. Originally a multi-class variable, it contained the values of 'NO', '>30' and '<30' that corresponds to a patient's readmittance days into the medical institution per encounter. However, in order to reduce the complexity of the classification problem, the feature has been re-coded into a binary problem such that Class-0 corresponds to 'NO' and '>30', while Class-1 corresponds to '<30'. With this newly coded arrangement, our binary target variable has the following distribution. As can be seen and was expected, we have quite severe data imbalance, which we will have to deal with in the further steps.

BEFORE:

	readmitted
NO	54864
>30	35545
<30	11357

AFTER:

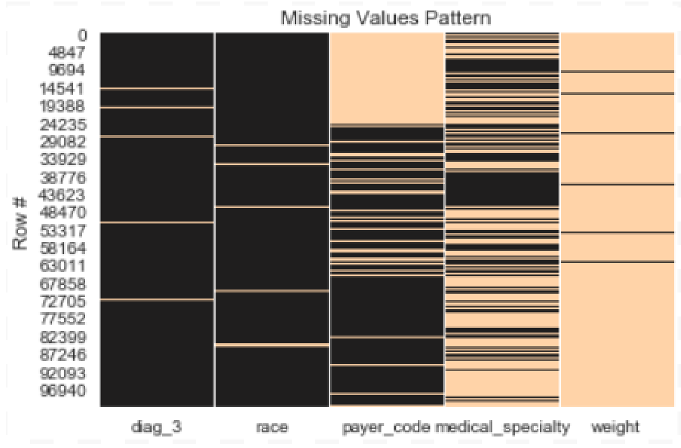
	readmitted
0	90409
1	11357

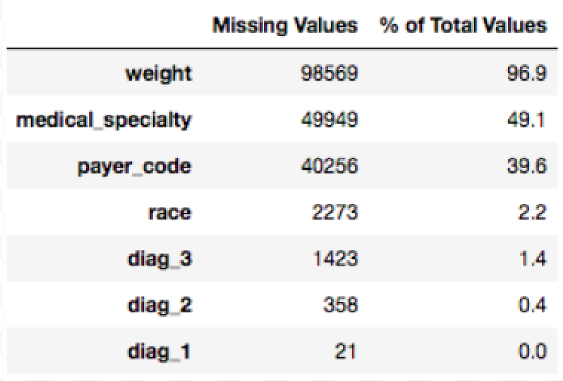


4. DATA CLEANING

4.1. Handling of Missing Values

We recognized that some of the features had missing values in it, and decided to look at the overall dataset to look at the distribution pattern and prevalence of them.





	Missing Values	% of Total Values
weight	98569	96.9
medical_specialty	49949	49.1
payer_code	40256	39.6
race	2273	2.2
diag_3	1423	1.4
diag_2	358	0.4
diag_1	21	0.0

We removed 3 features that contain large amount of missing values, which are:

- "weight", which has around 97% of values missing;
- "payer_code", which has around 40% missing values;
- "medical_speciality", which has around 49% missing values;

For Race, the percentage of missing values are relatively small, so we used mode imputation and replaced the missing values to it's largest category of "Caucasian". Similarly, we saw that Gender also has 3 missing values as 'Unknown', and took the approach of mode imputation here as well by encoding it as "Female". The handling of missing values in diag_1, diag_2 and diag_3 are discussed further down in section 4.2 Categorical Features.

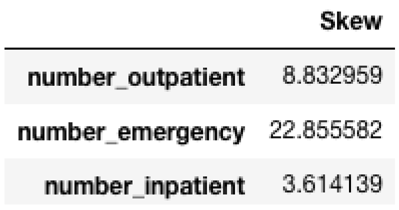
4.2. Numeric Features

We treated numeric features in two ways depending on the importance and value distributions:

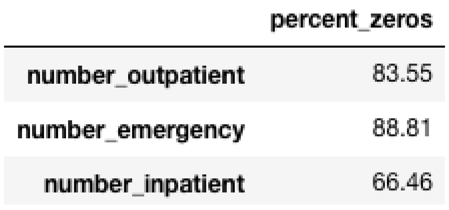
1) Apply Binary Coding where values greater than zero as assigned '1', else 0:
   - No. of Outpatient visits, No. of Emergency visits

2) Group values into set few categorical bins and then apply integer ordinal encoding:
   - Time in Hospital, No. of Lab Procedures, No. of Procedures, No. of Medications, No. of Diagnoses, No. of Inpatient visits

We decided to encode the numeric features in certain ways rather than retaining the original values because of the general nature of the dataset which is mainly categorical. While we took a bin-counting approach to ordinal encode majority of the numeric columns, we decided to treat the outpatient and emergency visits as ‘binary’ mainly because the majority value in these two columns were ‘0’, and they had high skew.



	Skew
number_outpatient	8.832959
number_emergency	22.855582
number_inpatient	3.614139



	percent_zeros
number_outpatient	83.55
number_emergency	88.81
number_inpatient	66.46

Although inpatient visits columns also had a majority value of 0, it can be observed that the relative skew compared to outpatient and emergency columns is less, and the percent of records are less comparatively as well. As a result, we treated inpatient visits like other numeric features and hence was encoded using bin-counting scheme. It is however important note that we also tried other treatment for numeric features such as retaining their raw input and so on, which is discussed in later sections on how we created multiple versions of datasets to determine the best performing version during our model fitting phase.

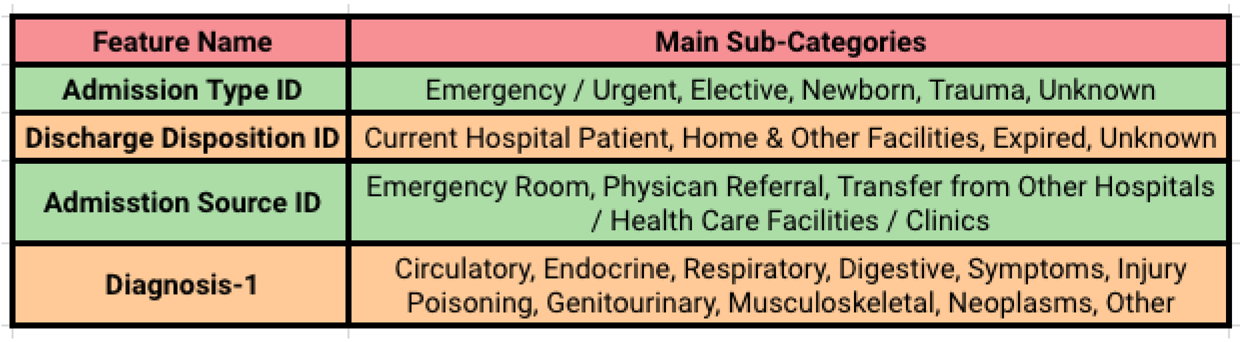
Another special treatment was applied for the Age feature. Although age is numeric by nature, it is grouped into 10 categories in the original dataset. In order to process the natural order of age and also not to make our data sparse, we applied bin-counting scheme by grouping age into 3 categories of ‘Youth’, ‘Adult’, and ‘Elderly’, which was later transformed through integer ordinal encoding.

4.3. Categorical Features

For categorical features, we employed three types of encoding techniques based on the nature of the data, which are: binary encoding, integer ordinal encoding and one-hot encoding.

A special treatment was applied for the 23 medications columns, where rather than following ordinal encoding, we decided to try out different encoding schemes to determine which gave the best results. Our main approach for the encoding was to the values in such a way that it represented {‘No’:0, ‘Steady’:1, ‘Up’:2 or 3, ‘Down’:2 or 3}. While the code for ‘No’ and ‘Steady’ remained the same, we tried interchanging the values for ‘Up’ and ‘Down’ because there are cases where reducing a medicine dosage could also require monitoring similar to when the dosage is increased. As a result, we believed that a modified version of ordinal encoding scheme would work the best when it comes to the 23 medications. Additionally, out of the 23 medications, only select 14 medications were retained based on their distributions.

When it comes to one-hot encoding, while some categorical features were straight forward, other columns had too many sub-categories in them, and hence we decided to collapse them further into smaller categories to prevent sparse columns.



Feature Name	Main Sub-Categories
Admission Type ID	Emergency / Urgent, Elective, Newborn, Trauma, Unknown
Discharge Disposition ID	Current Hospital Patient, Home & Other Facilities, Expired, Unknown
Admisstion Source ID	Emergency Room, Physican Referral, Transfer from Other Hospitals / Health Care Facilities / Clinics
Diagnosis-1	Circulatory, Endocrine, Respiratory, Digestive, Symptoms, Injury Poisoning, Genitourinary, Musculoskeletal, Neoplasms, Other

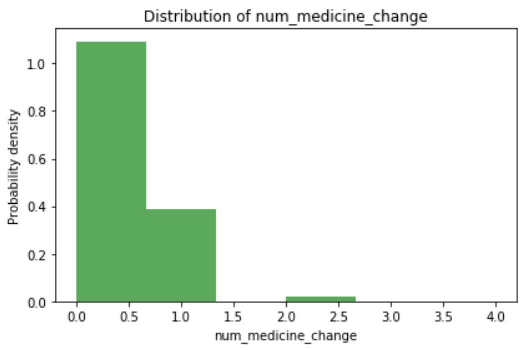
Additional time was spent on the three Diagnoses features, as they had vast number of levels compared to other features. We decided to keep only the diag_1 column and dropping the diag_2 / diag_3 as they has relatively more missing values, and had the potential of further complicating the dataset. We collapsed the diag_1 column into 9 main sub-categories based on ICD-9 codes (https://icd.codes/icd9cm), which corresponds to major primary diagnoses, and encoded the less prevalent values into a “Other” category. Since diag_1 also had few missing values into it, we coded them into the “Other” category as well. Similar approach was taken with other features such as discharge_disposition_id and so on, where we created additional categories such as ‘Other’ and ‘Unknown’ in order to capture all information.

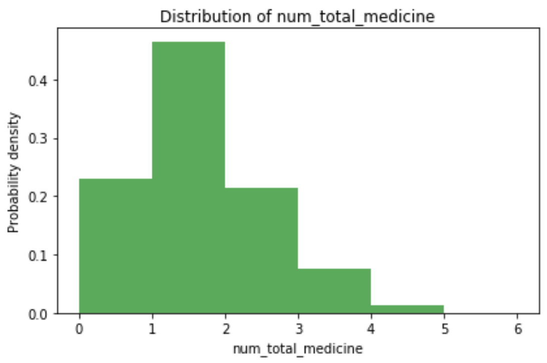
We applied binary encoding to features which are binary by nature, which includes response variable ‘readmitted’ and three binary features ‘gender’, ‘change’ and ‘diabetesMed’. For categorical features which are ordinal, we applied integer encoding to keep their natural orders. Those features includes: ‘A1Cresult’ and ‘max_glu_serum’. For categorical features which are nominal, we applied one-hot encoding. Some algorithms such as Logistic Regression are sensitive to the weights of values. Applying one-hot encoding of nominal features enabled us to reduce noise and error with these kinds of algorithms.

The features where the one-hot encoding technique was applied to are as follows: ‘race’, ‘admission_type_id’, ‘admission_source_id’, ‘discharge_diposition_id’, ‘diag_1’. Collapsing the vast amount of sub-categories into few select ones before applying one-hot encoding helped prevent generation of too many sparse columns and enabled us to retain only the most important information. For instance, for ‘discharge_diposition_id’, we collapsed the 30 types of discharge into 4 categories: home_other_facility, current_hospital_patient, expired and unknown, which, in turn, only led to 4 one-hot encoded columns rather than 30 separate columns.

4.4. Feature Engineering

As we can see from 3.2 Feature Analysis and Distribution, the 23 medication features are sparse with many columns highly skewed to a single value. In order to best discover the relationship of medication usage with readmission rate even after dropping few medications columns, we created two new features from the 23 medications:



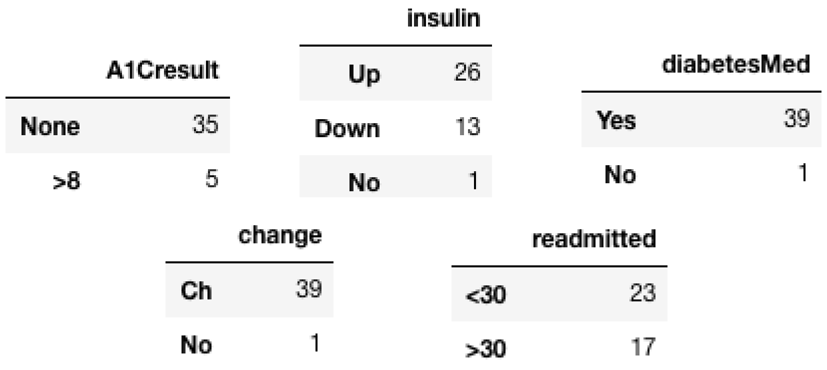


- ‘num_medicine_change’ counts the changes of medication usages during each encounter (visit), which was created by summing the ‘Up’ and ‘Down’ values for all 23 medications columns per row
- ‘num_total_medicine’ counts the total medication usages, which was created by summing the ‘Up’, ‘Down’ and ‘Steady’ values per row for the 23 medications

4.5. Creation of Multiple Clean Datasets

An early problem we noticed with this dataset is that each observation corresponds to one encounter (visit) instead of one patient. Some patients have multiple visits while others have only one visit. This means each observation may not be completely independent from the other. Given this, we had in mind two approaches to our dataset. The first and main approach (which we had named V2) was an instance based approach, in other words, based on the patient visits.

Another approach was based on the patients themselves and only retaining a unique patient record per encounter visit, in order to make the instances “independent” of each other, and better prepare the dataset for models such as Logistic Regression. Initially, we wanted to combine multiple patients instances into a single unit per patient by taking an “average” approach. However, we realized that this might lead to loss of important information per patient, especially with regards to the diagnosis, results and medications features. For example, below is the value distribution for certain features for a patient has had 40 encounters.



	A1Cresult
None	35
>8	5

	insulin
Up	26
Down	13
No	1

	diabetesMed
Yes	39
No	1

	change
Ch	39
No	1

	readmitted
<30	23
>30	17

While the demographics information such as Race/Gender/Age was same for all instances of this particular patient, there were changes in the values for test results, medications, and even the final readmission rate. If we had taken an “average” or “majority vote” approach, we would have missed the fact that the patient’s results for the A1C test had an abnormal value of above 8 for certain encounters, which would have affected our overall correctness of readmission rate as well. With this in mind, for our unique patient record approach, we decided to only retain the first encounter per patient in order to reduce noise in our dataset.

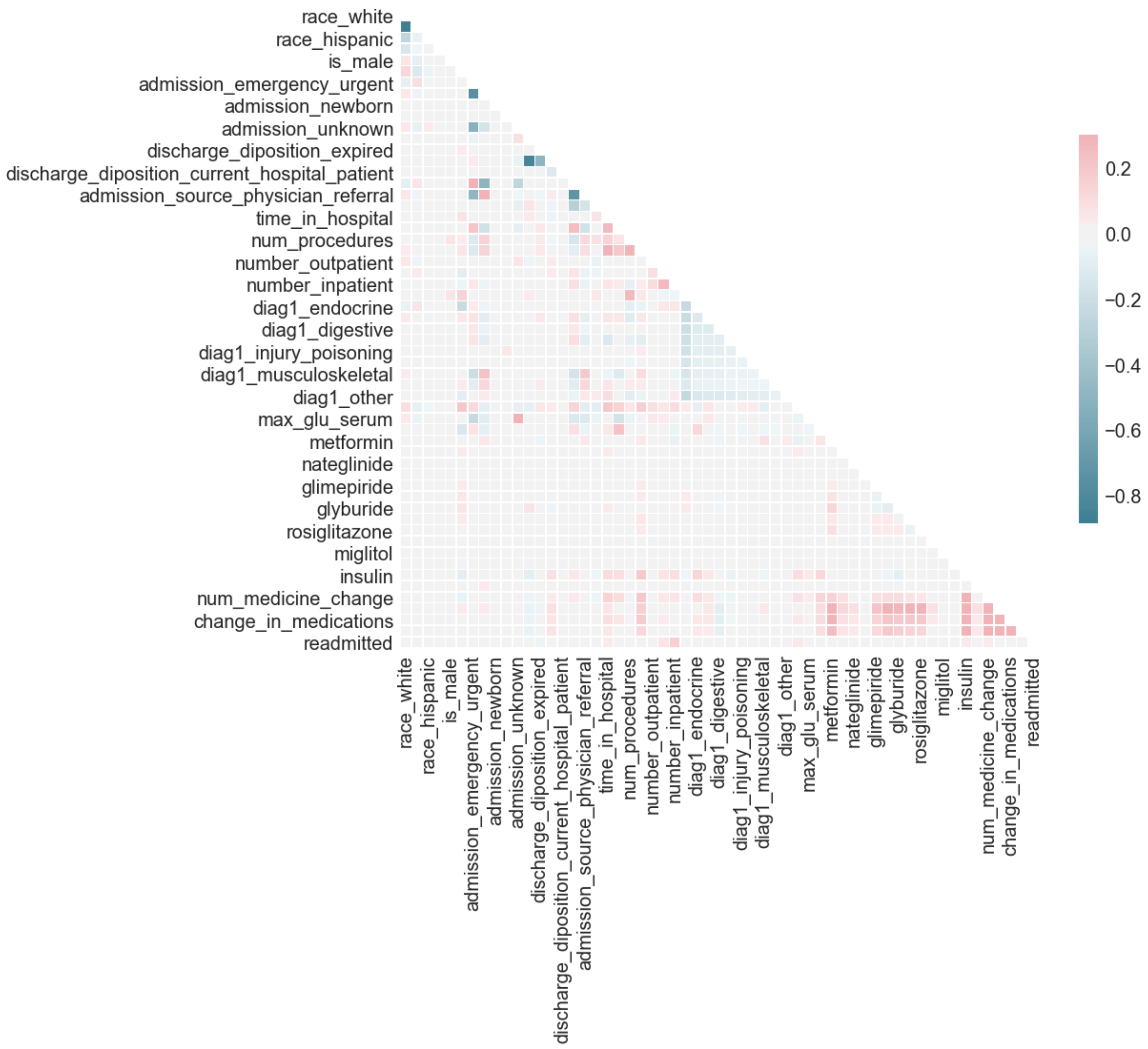
We created 7 different versions of dataset based on different assumptions and encoding rules. For example, in Dataset V5, we encoded {ICD-9 250.xx: diabetes} as a separate feature as we assume it might carry important information. We also removed all the observations which discharge_type correspond to 'expire' and 'hospice', as we assume those patients might have very small chances to be readmitted, so the distribution of this group is different from the rest.

In dataset V6, we kept all the numeric values as they are, instead of grouping and then integer / binary encoding some of them. Different encoding schemes for the medications columns were also applied, where in certain instances, we also tried retaining all 23 columns instead of only 14 of them. In some of the datasets we created, we decided to apply our unique patient approach and only have the first encounter per patient by removing rest of the records, which enabled us to truly see if the instances have a certain degree of independence between them based on the performance metrics of different versions of datasets.

After comparing the performance of different datasets, we decided to carry our analysis on dataset V2, as it has a relatively good performance, and we believe the structure of this dataset best represent the features. A detailed account on the performance of different datasets per model is discussed in the later section 5 Baseline Modeling.

4.6. Correlation Matrix

A correlation plot of our main cleaned V2 dataset, which has 101,766 instances and 56 features (plus one target variable readmitted), is displayed below. This shows that we don't have very strong correlation patterns among our features, though expected correlations among some variables like medicine and change_in_medications. We also realize that we should keep this correlation relation in mind while doing feature selections for our models with this dataset. Only certain levels of one-hot encoded columns for features such as diag_1, admisison_source_id etc., along with few select medications, are shown in the below plot for simplicity purposes.



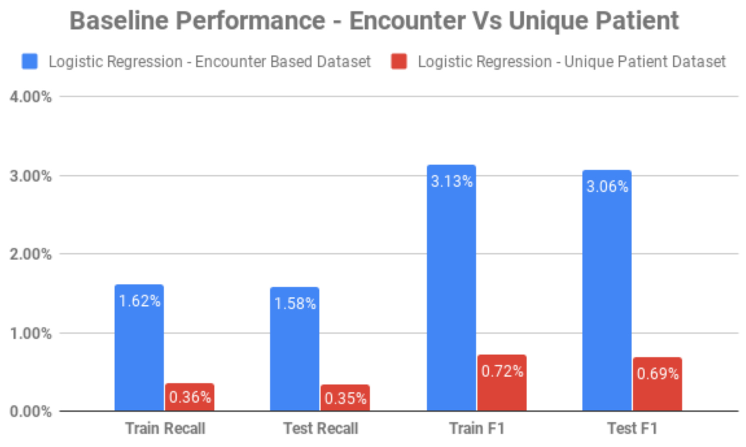
5. BASELINE MODELING

5.1. Baseline Approach to Compare Different Datasets

As mentioned previously in section 4.5, we created different versions of dataset by implementing varying cleaning and pre-processing techniques in order to capture the assumptions of the data. Although we took this approach, we also realize that it is imperative to have a “main” dataset to use for our models as well, especially for the later stages of parameter tuning and ensemble learning. In order to compare the performance metrics of our different datasets, we decide to use a baseline approach to assess the versions.

Our baseline approach consisted of using select machine learning classification models such as Random Forest, Logistic Regression etc. with their default settings and parameters to compare our datasets. We wanted to do a most basic trial run for this purpose, before even performing any pre-processing such as normalization / data balancing. This is mainly because we believed that if a particular version of dataset relatively doesn’t “perform well” during the basic trial run process, then it would be more difficult to bring up the relative metrics during the later stages of the model fitting. For example, among 3 versions of datasets, if there is an apparent increased performance in one of the versions during the baseline trial run, then it can be assumed that it will a relatively increased performance as well in later stages.

We of course kept in mind that there is possibility we might have to go back to another version of dataset should we reach a dead-end with the current one. The baseline approach also enabled us to see the difference in performance between an encounter-based dataset and an unique-patient based dataset, given that multiple patients had more than one encounter. The below graph highlights the difference between the Train/Test Recall and F1 Score for these two datasets with Logistic Regression model, and it becomes apparent that encounter-based approach is a better choice comparatively. The following sections provide a detailed account on our model implementation techniques and results for our main V2 encounter based dataset.



6. NORMALIZATION & DATASET BALANCING

6.1. Normalization

Normalization is often applied to datasets upon applying statistical models to them. Normalization subdues the effects of differing scales that different ranges can have. In our data, after cleaning and coding, we don't have major problems with such scales in the features. In fact, most of our data has either one-hot encoding (binary) or discrete numbers coded to mean a certain attribute. Yet, it is still appropriate to apply normalization.

There are a few normalization techniques that can be applied, we used MinMax and StandardScaler scaling. MinMax scaling in short transforms the data into a fixed range between 0 to 1. Given our data already has small standard deviations, and given negative values or very extreme data points were not playing a role, this approach made more sense for our purposes. That being said, we took into account one disadvantage of MinMax scaling is that the effect of outliers are suppressed. The Standard Scaler removes the mean and scales to unit variance. We found that performance at times varied when these two were applied.

We also made sure that fit and transform only our train data, and then use the training data's "fit" to transform the test data. This was done to prevent any data leakage, since in practice, upon having unseen data, you will be transforming with the characteristics of the data that you already have.

How we implemented it in the code:

```
scaler = preprocessing.MinMaxScaler() or StandardScaler()
train_X = scaler.fit_transform(train_X.astype(np.float64))
test_X = scaler.transform(test_X.astype(np.float64))
```

6.2. SMOTE

As mentioned earlier, class imbalance is a major issue in this dataset, albeit a natural problem in medical data. SMOTE is a sophisticated algorithm that in essence creates carefully constructed "artificial data points" in order to bring a balance between the two predictive classes in the training phase. It is important to emphasize that we only apply SMOTE at the training phase. It does not make sense, for example, to apply SMOTE to the test data which we are trying to predict. It in a way would be like "cheating" if the data you are wanting to predict is augmented with artificial data points. We simply want for our model at training to have a balanced fit so that it is as robust as possible for unseen data. Another advantage with SMOTE was that we did not lose data but augmented data in a smart manner. A disadvantage that we noticed was higher computation time given increased data and also increase in our variance.

To demonstrate the effects of SMOTE, we ran our candidate models both with and without SMOTE, and as well with Under Sampling technique. As an example, if our data was split into 80% train and 20% test, SMOTE would augment our predictive classes of the training data in the following manner:

	Class 0	**Class 1**
Without SMOTE	72268	9144
With SMOTE	72268	72268

One important note that we must mention about our SMOTE implementation is that SMOTE has been applied **after** our data has been split in the cross validation. This is to prevent any SMOTE leakage into our test set, which we want to remain clean. We do not want SMOTE fitting with the test data and were careful to make sure this step was done after the train/test split.

Another important thing about SMOTE that should be stated is that the SMOTE cannot take in text data points and only works with numeric data. This is understandable because such text data cannot be “artificalized”. This fact was taken into consideration during our data cleaning process. We though best simply not to have any text data for such cases, especially with SMOTE, which was another one of the main reasons why one-hot encoding was implemented.

SMOTE however did not always bring about better results. We noticed that with certain models and runs, SMOTE would increase bias, and accuracy would suffer more. Thus, we were careful where we used it and where we didn’t by evaluating different combinations on dataset.

6.3. Under Sampling

Another technique for balancing classes, which we used, is under sampling. Under sampling simply samples the given data to a lower volume, until a balance in classes is achieved. Throughout the testing of our models, we tested with under sampling often. While the effect of undersampling will be more clear in the results tabulations, one thing that can be said is that, in general, under sampling stabilized the gap in metrics between test and train. However, often times, under sampling lead to lower accuracies and recalls. This is expected, because with the much lower amount of particular class, under sampling leads to a much lower amount of data to train on, thus lowering the predictive accuracy.

Below is how our training data looks with and without under sampling. Similar with SMOTE, Under Sampling was only implemented on the train data, while the test was untouched:

	Class 0	**Class 1**
Without Under Sampling	72268	9144
With Under Sampling	9144	9144

As is evident, we can see that there is much less data to work with than with SMOTE. The tradeoff being that at least the less data is “real” data, in contrast to SMOTE where despite having more data, it is inflated with artificial points.

7. PARAMETER TUNING

7.1. Grid Search and Randomized Grid Search

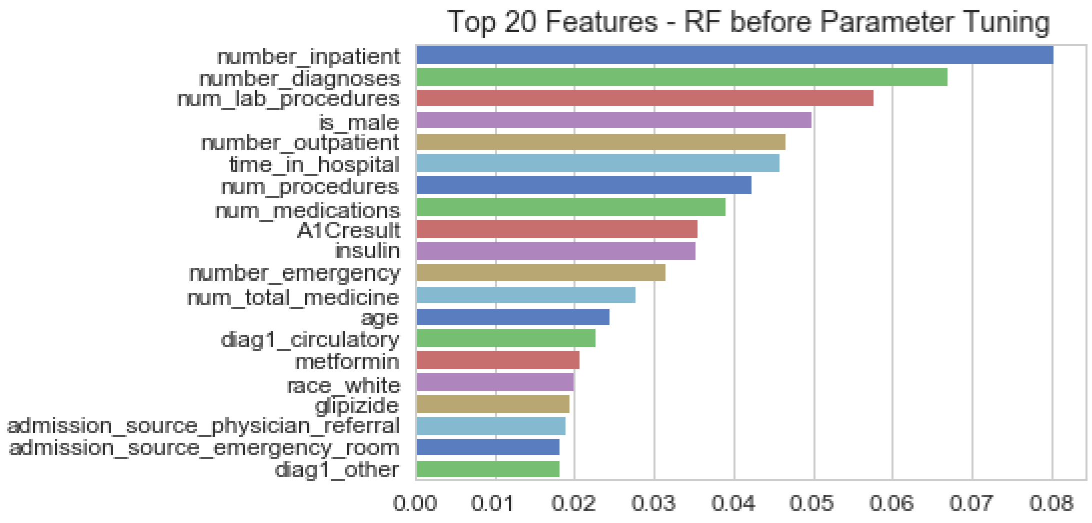
For tuning our parameters properly we utilized the GridSearch utility in Python. GridSearch is very convenient in that it allows the user to focus on which parameters for a given model should be searched. Importantly, it allows one to do that search while optimizing for certain attributes. Given our dataset’s heavy class imbalance, we knew from preliminary modelling that the bigger issue was with the performance of classifying class 1. Hence, what we aimed for was to have balanced accuracies between the two classes and of course high recall. With GridSearch it was possible to optimize for these metrics and thus receive a model that would fulfill our desired optimizations. After learning about this capability, we began to tune all our models with GridSearch, specifically scoring for 'recall' and 'balanced_accuracy'.

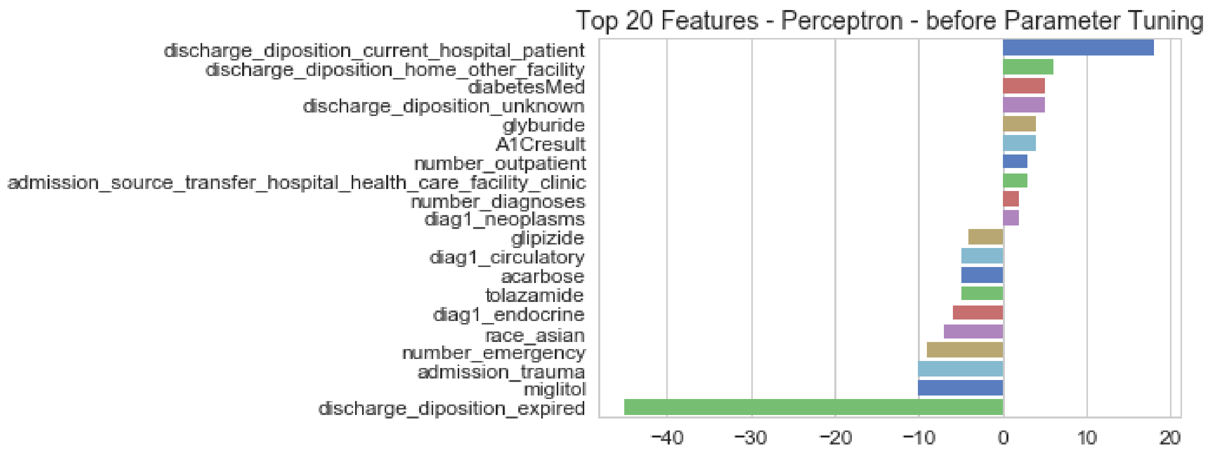
One drawback with Grid Search is that it could get very computationally expensive as the number of parameter combinations for each model increases, as it follows a ‘brute-force’ approach to find the best parameter set. As a result, we decided to utilize the Randomized Grid Search utility also available in Python that ‘randomly’ tried a specified number of combinations that gives the best metrics. This helped us narrow down certain parameter ranges for our models, especially for integer/numeric based parameters, allowed us to remove those parameter values from our combinations that brought down the performance metrics significantly. This trial-and-error approach for parameter tuning enabled us to save resources while finding the optimal set. A detailed account on the effects of parameter tuning for different classifications models are mentioned in the below sections.

8. FEATURE SELECTION

8.1. Methodology for Determining Feature Importance

When it comes to optimizing machine learning models, feature selection is just as important as parameter tuning to achieve the 'best' performance metrics. In order to get an overall picture of the features in our dataset, we used different models' feature importances or coefficient parameter to assess the relative relationship among the features. We observed that the 'level' of importance for the features varied based on the models as well as the parameter settings. Additionally, the number of features with 'zero' importance also changed model to model. Below are plots of features importance for Random Forest and Perceptron (baseline run on train data), that shows the 'top 20' features for simplicity purposes. It can be noted that while Random Forest gives a score of relative importance to the features, Perceptron uses the direct coefficient score for the actual features, due to which many had negative or zero coefficients.





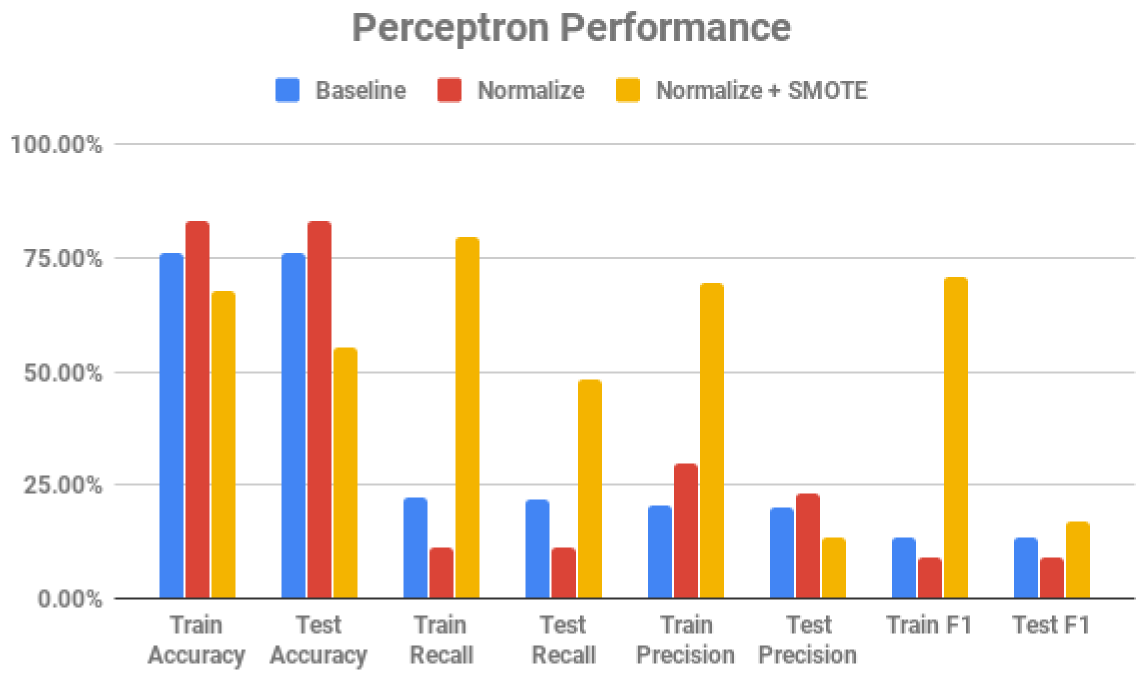
Another approach to feature selection, was using Principal Component Analysis, in essence, to see which features are contributing the most (loadings) given a reduced dimensionality space. The way we assessed PCA's feature importance was seeing which features had close to zero loadings for the primary principal components.

While these methods were utilized for a more "big picture" view of our features and their relevances, for our models specifically, we utilized the filter method for selecting features. For some models like Linear SVM, the feature engineering was not so important and we understood that in the more expansive ensemble models one had to test the effect of feature engineering on the overall performance.

9. MODEL COMPARISONS & EVALUATIONS

9.1. Perceptron

Perceptron is a simple linear classification model that works best for linearly separable data by converging at certain point. We decided to try out Perceptron model in order to verify if there is any linear relationship present in our dataset and to see if can be linearly separable. The below plot compares Perceptron performance with data pre-processing techniques such as Normalization and SMOTE.

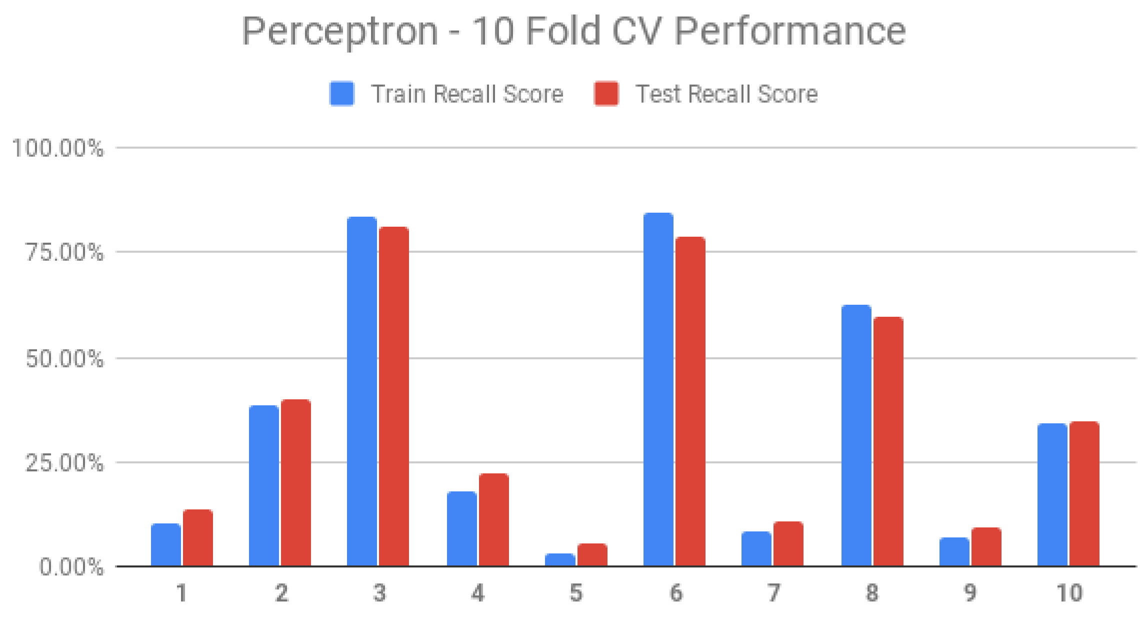


It can be observed that while the train/test accuracy decreased, the recall score increased tremendously after taking care of imbalanced data using SMOTE technique. However, at the same time, it can be noted that there are cases of over-fitting between train and test data where the recall metrics is the highest. After trying out Parameter Tuning with different combinations of values, it was found that the below default parameters give out the best results:

```
Perceptron(alpha=0.0001, class_weight=None,early_stopping=False,eta0=1.0,
     fit_intercept=True, max_iter=None, n_iter=None, n_iter_no_change=5,
     n_jobs=None, penalty=None, random_state=0, shuffle=True, tol=None,
     validation_fraction=0.1, verbose=0, warm_start=False)
```

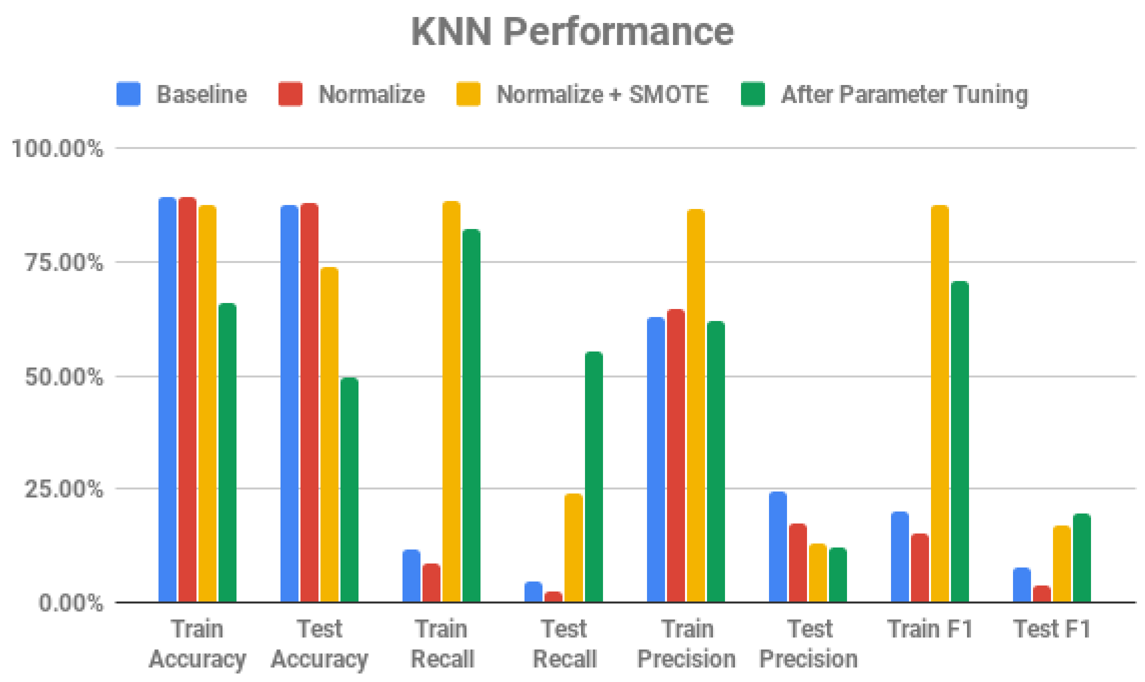
Similarly, since Perceptron internally takes care of the feature selection process using coefficient methodology, removing features tended to decrease the overall performance, because of which we decided to keep all the features for Perceptron. The best results for the Perceptron are with the default parameters with the dataset treated to Normalization and SMOTE techniques, whose metrics are displayed in the plot above. As a result, it can be seen that Perceptron over-fits our dataset, further implying our data could not be linearly separable.

Another important thing that was noted during the model fitting process, in addition to its overfitting tendencies, is that there were inconsistencies with the metrics during our 10-Fold Cross Validation process. For example, below plot shows the train/test recall scores for a typical run of the Perceptron model. It can be observed that there is drastic changes in the scores per iteration, with only couple of those having 'high' metrics. This suggests that the overall score that is obtained through 10-Fold CV, which is the 'average' of all runs, is heavily influenced by these few runs that 'drag' the score to be higher than it really is.



9.2. KNN

The KNN model, also known as K-Nearest Neighbors, is a simple and easy to understand classification algorithm that does not have prior assumptions about nature and distribution of the data, which makes it a straightforward and lazy model. We chose KNN model because of its ability to 'quickly' determine relationships between instances, which would enable us to get an idea on possibility of 'neighbor' based predictions for our dataset. The below plot gives a summarized result on various stages of the KNN model fitting process:



The result plot conveys the fact that preprocessing with Normalization/SMOTE helped increased the train/test recall score compare to its relative baseline performance. By looking at the current plot and that of Perceptron in the previous section, it becomes apparent that there is a tradeoff between the Accuracy - Recall score, especially visible by decrease in accuracy value when the recall score is increased.

We can also observe that Parameter tuning led to increase in Test Recall value, almost doubling it, but decreased the value of accuracy further down. We can also note a decrease in the Precision value as we optimize our model for increased recall score, which can be explained by the imbalanced nature of the dataset and the 'additional' artificial class-1 points that were created through SMOTE to address the same. During the parameter tuning phase, we noticed that Recall as well as F1 scores tended to increase as we increased the value for the number of neighbors parameter, which suggests that the model 'learned' better as more data points were supplied.

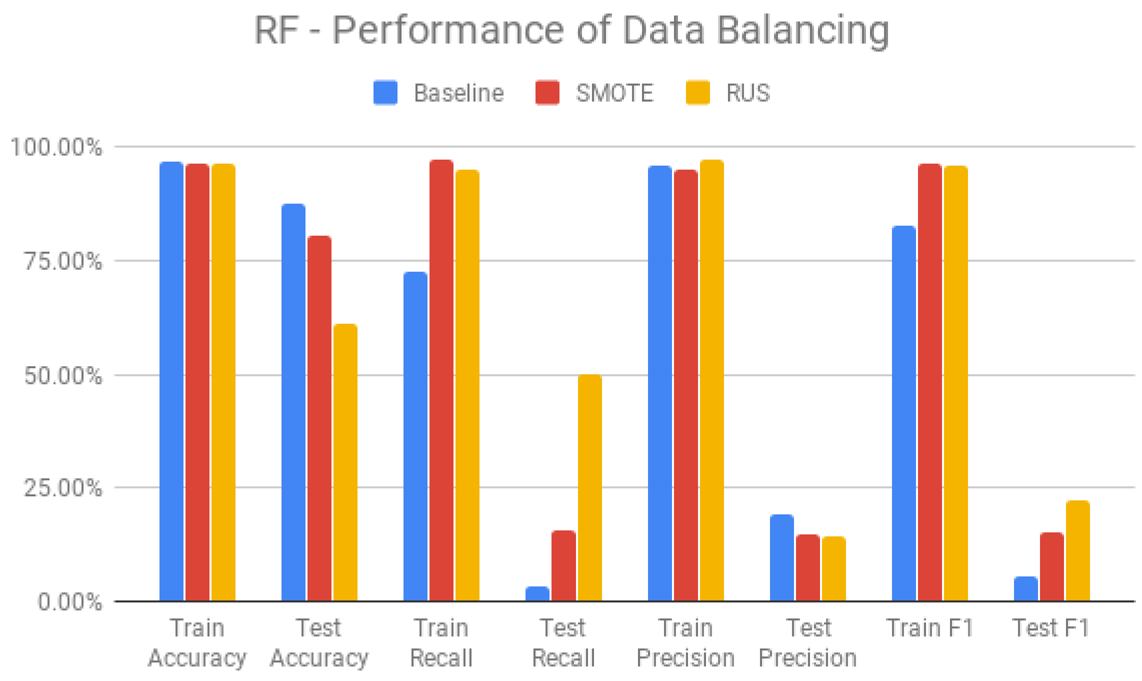
The optimal parameter set for KNN is as follows:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=401, p=2,
                     weights='uniform')
```

One major disadvantage that we observed with KNN is that it is very computationally expensive, which made our CV runs take longer time than expected, especially with the overhead of applying SMOTE during each iteration. Given the fact that our dataset had 50+ features, the "curse of dimensionality" could also have been a contributor to the extended runtime. As a result of the extensive resources required for each run, we decided to not do feature selection, and instead use the model results for gaining a better understanding of our dataset.

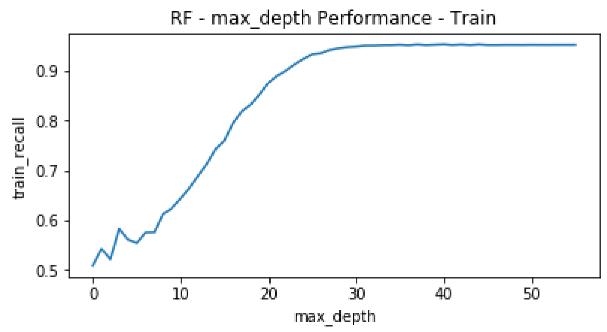
9.3. Random Forest

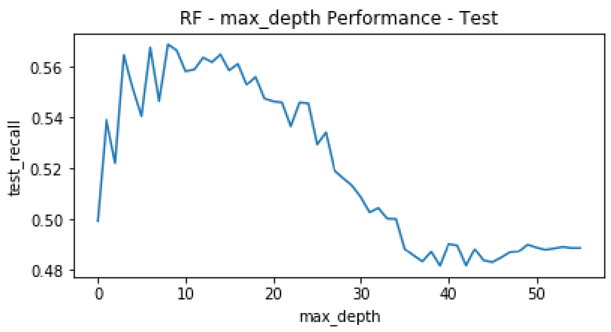
Random Forest is a sophisticated ensemble based classification algorithm that makes predictions by building decision trees. We decided to try out Random Forest directly rather than modeling with decision trees because of this model's capability reduce bias in the data. The below plot summarizes the result of data balancing techniques applied with Random Forest:



While SMOTE worked well for the previous two models of Perceptron and KNN, it is clear than Random Under Sampling technique helps bring up the recall and f1 score when it comes to Random Forest. This might be because Under Sampling balances dataset without adding in any artificial data. From the result plot above it can also be observed that is again a tradeoff present with the accuracy and recall, and that accuracy score gradually decreases as we try to bring up the recall/f1 score of the model. It is important to note that the model produced similar result with and without Normalization. However, we decided to apply normalization for consistency purposes.

The next step in the Random Forest model fitting is parameter tuning. Unlike the first two models, Random Forest has many important parameters that needs to tuned in order to find the optimal set that produces the best results. Some of the notable parameters that affects the performance are: max_depth, n_estimators, max_features, min_samples_leaf and so on. Each of the parameters help determine the extent to which a decision tree within the Random Forest must be built, while also controlling the height and number of trees within the model. The parameter combinations were assessed through Grid Search optimized for recall and balanced accuracy.

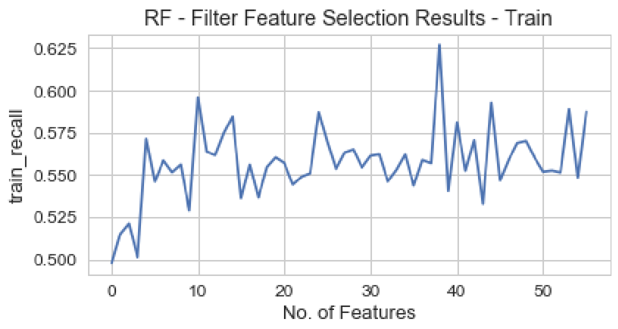


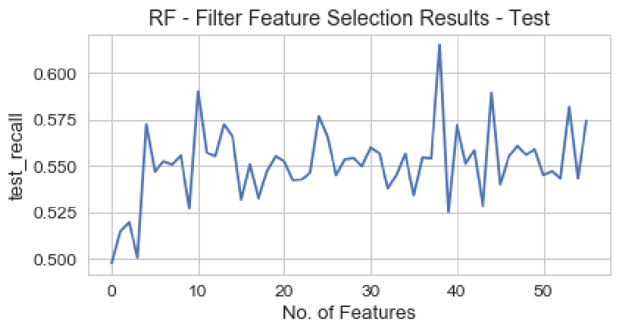


The above plot is the performance of one such parameter of Random Forest for different values. It can be seen that as the max_depth of the decision trees within the Random Forest Model increases, the train recall keeps increasing till it reach a maximum value, after which it remains the same throughout. On the other other, fluctuations in the test recall value can be seen in the adjacent plot. It can especially be observed that for a certain range of max_depth value, the test recall tends to keep decreasing. Such a approach was taken with all other parameter as well to determine the increase/decrease in performance metrics for parameter values. After assessing the values of all possible combinations for Random Forest parameters, below is the optimal set that produces best recall as well as balanced accuracy score:

```
 RandomForestClassifier(bootstrap=True, class_weight='balanced_subsample',
             criterion='gini', max_depth=8, max_features='log2',
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=340,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
n_estimators=10, n_jobs=None, oob_score=False,verbose=0, warm_start=False)
```

The subsequent step in the model fitting process is Feature selection, which produced interesting results compared to those of previous models such as Perceptron. The internal feature ranking system of Random Forest enabled us to run a filter method for feature selection, whose results are displayed in the below plot. It can seen that both train and test recall attain a max score at a particular number, which is top 39 features of the model in this case.

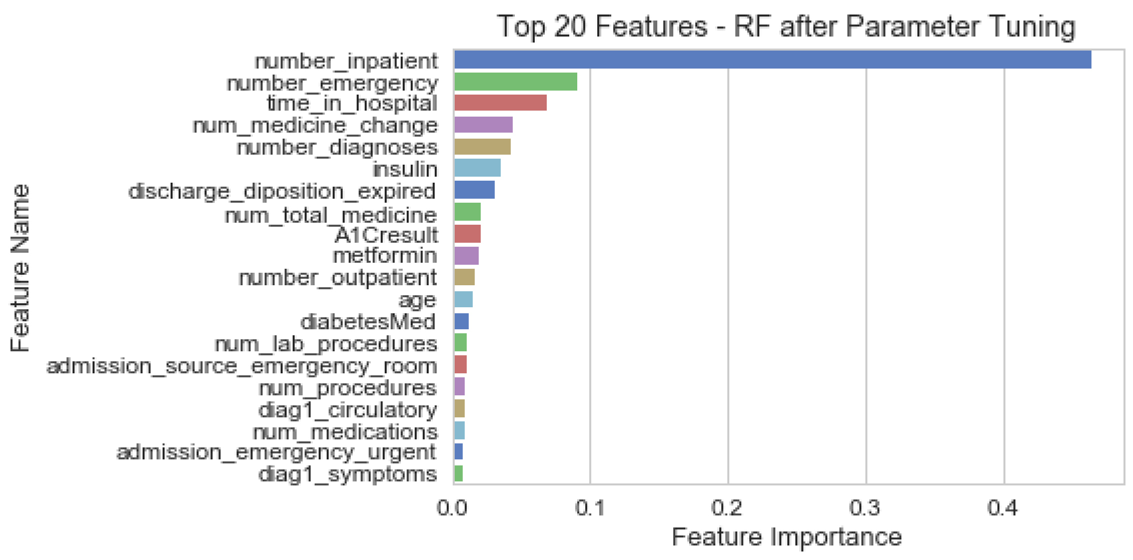




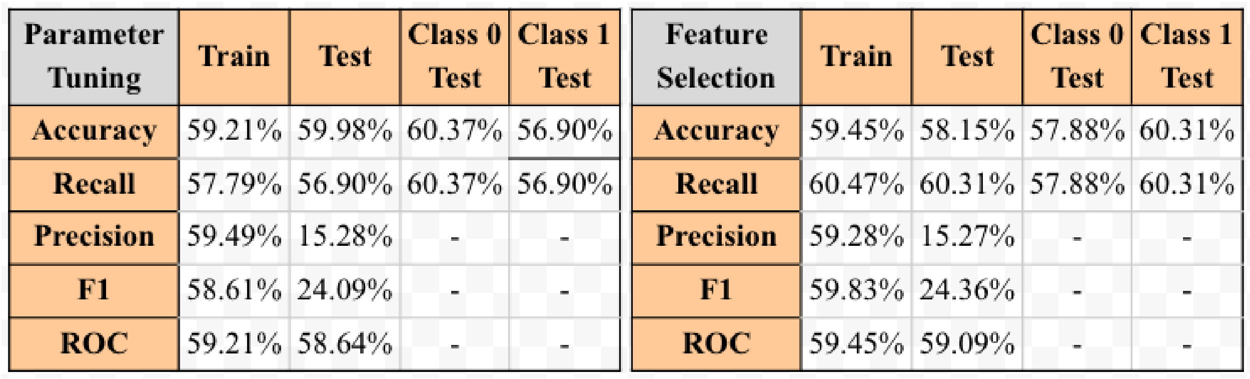
It is important to mention however that filter feature selection was only performed after parameter tuning utilizing the parameters obtained above for Random Forest. It is interesting to note that Random Forest produced contrasting results for features importances depending on the parameters used. For example, before parameter tuning, there were no ‘zero’ valued feature importance, however, after parameter tuning, there were 14 features whose relative importance was marked as zero. Those features are:

index	importance
tolazamide	0.0
race_hispanic	0.0
race_asian	0.0
admission_newborn	0.0
glyburide_metformin	0.0
admission_trauma	0.0
diag1_genitourinary	0.0
miglitol	0.0
acarbose	0.0
chlorpropamide	0.0
nateglinide	0.0
repaglinide	0.0
discharge_diposition_current_hospital_patient	0.0
discharge_diposition_unknown	0.0

Additionally, the feature selection process with Random Forest after Parameter Tuning also changed the order of importance for the top features. Below is a plot for Top 20 Features Importance of Random Forest after parameter tuning process:



The above plot aids with determining the influential features that affect the readmittance rate after a particular encounter of a patient. It can be seen that some of the diabetes related features are in the top such as insulin, A1C Result, diabetes Med etc., along with important hospital metrics. In addition to providing an insight, the feature selection process also helped increase the performance metrics from what was found in Parameter Tuning.



Parameter Tuning	Train	Test	Class 0 Test	Class 1 Test
Accuracy	59.21%	59.98%	60.37%	56.90%
Recall	57.79%	56.90%	60.37%	56.90%
Precision	59.49%	15.28%	-	-
F1	58.61%	24.09%	-	-
ROC	59.21%	58.64%	-	-

Feature Selection	Train	Test	Class 0 Test	Class 1 Test
Accuracy	59.45%	58.15%	57.88%	60.31%
Recall	60.47%	60.31%	57.88%	60.31%
Precision	59.28%	15.27%	-	-
F1	59.83%	24.36%	-	-
ROC	59.45%	59.09%	-	-

The above table provides comparison on the performance metrics between the two steps. It can be seen that the recall score along with f1 increased after feature selection, while both Class 0 and Class 1 metrics are above 50%, although Class 0 metrics decreased slightly from Parameter Tuning phase. However, this was expected, as there has been a trade-off between Class 0 and Class 1 performance throughout the model fitting process.

9.4 Naive Bayes

Naive Bayes is a simple but powerful algorithm that is widely used for analyzing medical data. It is built on the assumption that each feature on a given class is independent from the others. As discovered in 4.6 Correlation Matrix, we don't see strong correlation patterns among our features, although some correlations are expected. From 2.2 Feature Analysis and Distributions, we discovered that most of the feature values are not normally distributed and we are more inclined to believe they are multinomial. We tested the performance with both distributions to validate our assumptions.

As we can see in the baseline case (without normalization and SMOTE), neither Gaussian or Multinomial Naïve Bayes produced balanced and accurate results.

Gaussian Naïve Bayes - Baseline Performance

BASELINE	Train	Test	class_0_test	class_1_test
Accuracy	13.10%	13.08%	2.21%	99.65%
Recall	99.77%	99.65%	2.21%	99.65%
Precision	11.36%	11.35%	-	-
F1	20.40%	20.37%	-	-
ROC	50.99%	50.93%	-	-

Multinomial Naïve Bayes - Baseline Performance

BASELINE	Train	Test	class_0_test	class_1_test
Accuracy	88.81%	88.81%	99.95%	0.15%
Recall	0.17%	0.15%	99.95%	0.15%
Precision	28.07%	28.50%	-	-
F1	0.33%	0.30%	-	-
ROC	50.06%	50.05%	-	-

After parameter turning, we found the best performance occurred when we apply MinMax Scalar and SMOTE for multinomial with the following parameters:

```
MultinomialNB(alpha=0.5, class_prior=None, fit_prior=False)
```

10-Fold run of our chosen Multinomial model with SMOTE yields:

BEST CASE	Train	Test	class_0_test	class_1_test
Accuracy	59.06%	60.60%	61.19%	55.92%
Recall	56.94%	55.92%	61.19%	55.92%
Precision	59.46%	15.32%	-	-
F1	58.17%	24.05%	-	-
ROC	59.06%	58.56%	-	-

9.5 Logistic Regression

Logistic regression is built on the assumption that observations are independent from each other. In our case, we tested logistic regression on both dataset V2 of unique encounters and dataset V5 of unique patients. We found in the baseline case, the performances on those two datasets are similar. Thus, we decided to build the model on V2 incoherent with other models. In the base case on V2, logistic regression gives the following performance:

Logistic Regression - Baseline Performance

BASELINE	Train	Test	class_0_test	class_1_test
Accuracy	88.81%	88.81%	99.95%	0.15%
Recall	0.17%	0.15%	99.95%	0.15%
Precision	28.07%	28.50%	-	-
F1	0.33%	0.30%	-	-
ROC	50.06%	50.05%	-	-

With Gridsearch, we selected optimal parameters based on the best recall and balanced accuracy rate. The optimal parameters occurred as:

```
LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='newton-cg',
          tol=0.0001, verbose=0, warm_start=False)
```

10-Fold run of our chosen Logistic Regression with standardization and SMOTE yields:

BEST CASE	Train	Test	class_0_test	class_1_test
Accuracy	61.18%	63.47%	64.53%	55.01%
Recall	57.80%	55.01%	64.53%	55.01%
Precision	61.99%	16.30%	-	-
F1	59.82%	25.15%	-	-
ROC	59.82%	25.15%	-	-

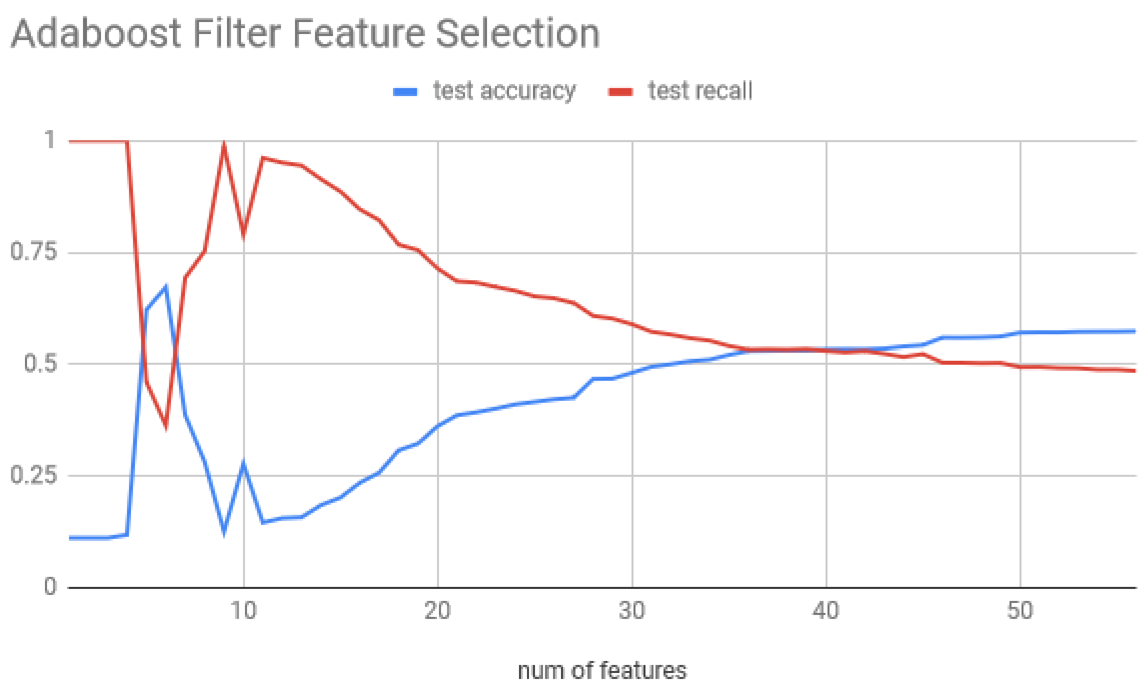
9.6 Adaboost (with Decision Tree)

Adaboost has an ensemble like approach to learning as it fits a sequence of weak learners on repeatedly modified versions of the data, then using a weighted majority vote to reach a final prediction. This approach can be very robust and its for the reason this was one of the algorithms we chose to explore. The weak learner chosen for Adaboost was Decision tree. We used grid search to obtain the optimal parameters for our decision tree.

We noticed that adaboost performed best at its baseline; in other words, without sampling or normalization. Here are the 10-Fold Results:

BASELINE	Train	Test	class_0_test	class_1_test
Accuracy	68.06%	57.08%	58.16%	48.50%
Recall	100.00%	48.50%	58.16%	48.50%
Precision	25.89%	12.71%	-	-
F1	41.13%	20.14%	-	-
ROC	82.02%	53.33%	-	-

We also ran a filter feature selection for the adaboost model looking at both test recall and test accuracy:

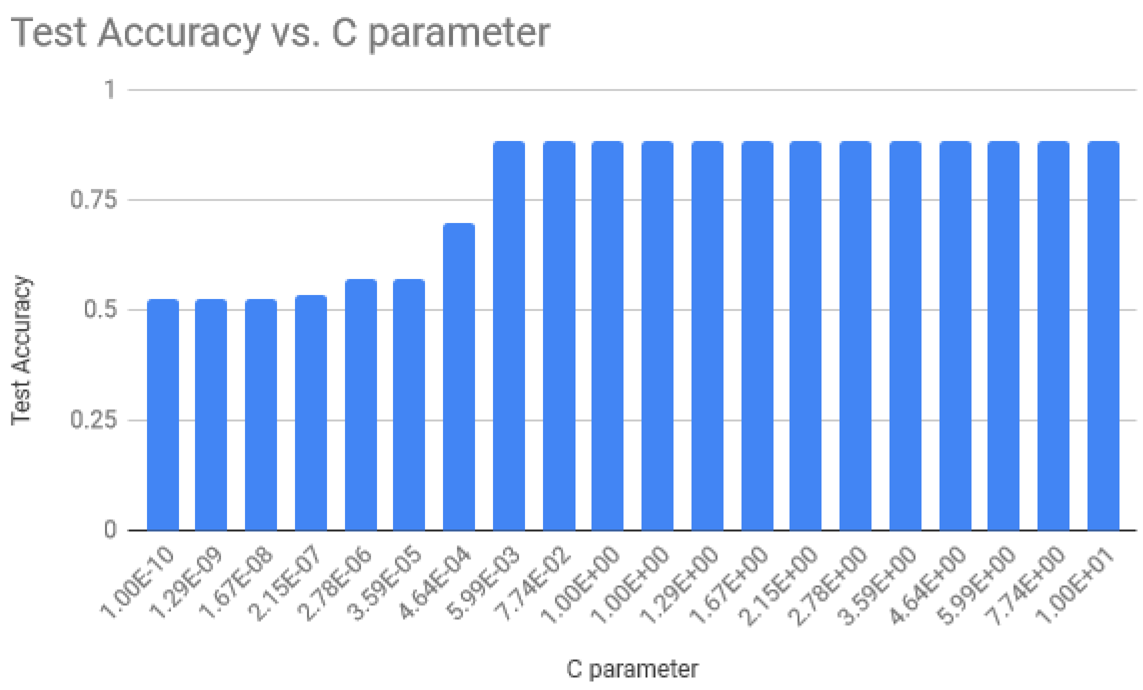


We notice that with higher features the accuracy and recall is stabilized slightly above .5, as before there is a large gap and quite a large amount of volatility. From this, we can either choose 40 features and have essentially equal accuracy or recall, or take in all features, have higher accuracy and a bit lower recall. For the time being, we decided to keep all features.

9.7 SVM

The SVM algorithm is a very robust model that is a maximum margin classifier and is able to efficiently adjust to different data structures. In implementing the SVM algorithm, it was first imperative to determine which kernel to use. Three choices we had: linear, polynomial, or rbf. Initial runs on polynomial showed very low performance and runs on rbf kernel took very long computation time, too long to be feasible for use. The rbf kernel results were average with quite a bit of overfitting even after adjustment of regularization parameter. This left us with the understandable choice of implementing a linear SVM.

In choosing the C parameter for the linear SVM model, a typical range was searched via Grid Search. The recreated resulting accuracy:



It was apparent to us that we reach a plateau at accuracy of .88 and thus can simply use 1 as the C parameter, as GridSearch suggested to us as the optimal C.

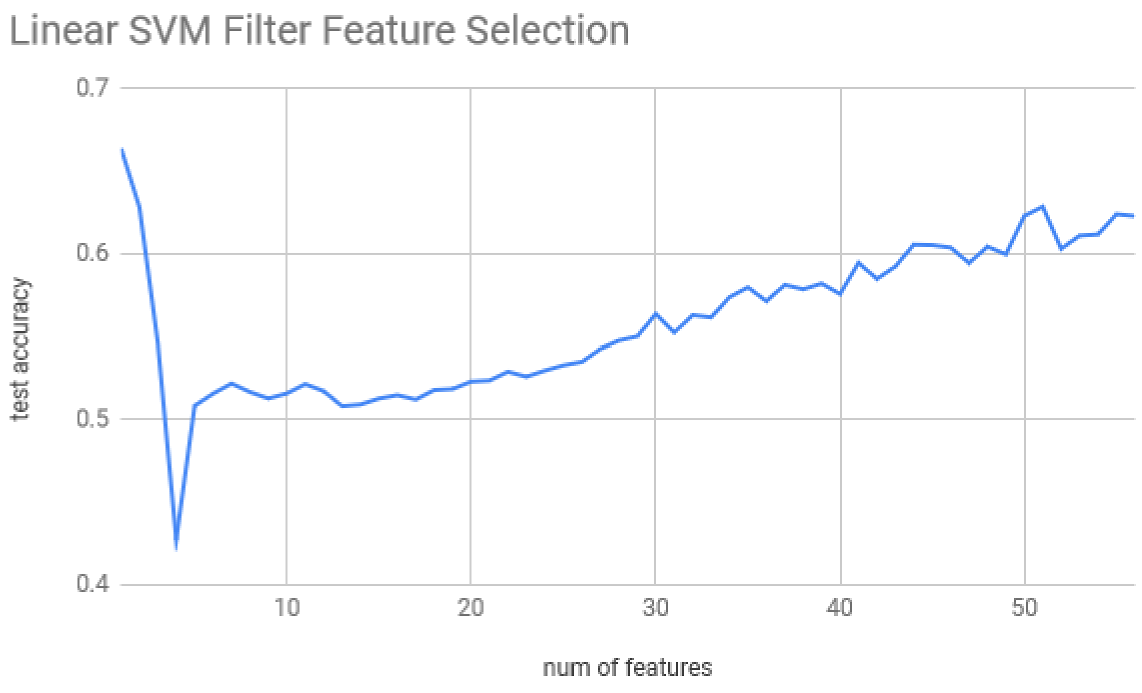
We tuned for the parameters via Gridsearch and we optimized for recall and balanced accuracies in order to yield a model that would optimize class 1 accuracy and recall in conjunction with class 0. An important parameter that we set for SVM to find optimal solution was the class weights, because it was expected we would need bigger weights for class 1 given the imbalance.

Given our optimizations, gridsearch yielded as the best Linear SVM model:

```
LinearSVC(C=1, class_weight={1: 10}, dual=True, fit_intercept=True,
      intercept_scaling=1, loss='squared_hinge', max_iter=1000,
      multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
      verbose=0)
```

It should be noted that the RBF kernel was also tested, but it's very heavy computation time made it impractical for further use. The polynomial kernel, expectedly, was not at all good at picking up signal.

While feature selection is not typically a big concern for this model, especially given the weight assignments, we decided nevertheless to run the filter method to see how the number of features (ranked by highest correlated) contributed to test accuracy:



As one can see there is a large test accuracy point with the first feature, and after a drop the test accuracy slowly grows to almost where it began. This feature selection does not convey any definite approach to how many of the top correlated features we should have, it seems best to keep all features, especially given the robust nature of the SVM model.

10-Fold run of our chosen Linear SVM model with no SMOTE yields:

	train	test	class_0_test	class_1_test
accuracy	0.618286	0.618890	0.62629	0.559893
recall	0.560280	0.559893	0.62629	0.559893
precision	0.158290	0.158492	-	-
f1	0.246793	0.246981	-	-
roc	0.592926	0.593092	-	-

We are happy to see stabilization between train and test among the recall and accuracy, also we notice that class 1 recall/accuracy is above .50 (given definition the conditional recall/accuracy is equal to each other). That being said, we think there is potential to do better and raise both the accuracy and recall.

For comparison sake, let us see our model performance with SMOTE:

	train	test	class_0_test	class_1_test
accuracy	0.509468	0.129994	0.0210476	0.997233
recall	0.998037	0.997233	0.0210476	0.997233
precision	0.504789	0.113449	-	-
f1	0.670467	0.203714	-	-
roc	0.509468	0.509140	-	-

We see that SMOTE introduces a great amount of bias, and overfits our recall. We believe that this Linear SVM model without SMOTE is the most robust and generalizable and thus remain with that model, but seek other models to either improve with it or simply be better model than SVM is as we have it.

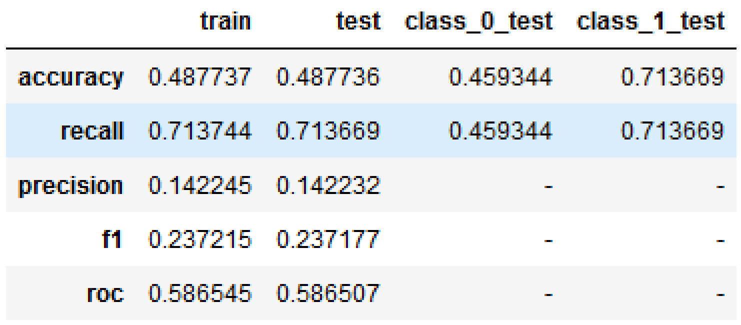
9.8 Gradient Tree Boosting

In class, we learned about gradient loss as it is applied to regression. The gradient boosting algorithm builds upon that notion as it builds an additive model based upon weak learners (regression trees) which are fit on the negative gradient of the binomial or multinomial deviance loss function. Given our supposition of tree based classifier working well on this data, we decided to employ this model.

From GridSearch, optimizing for the respective scoring metrics, balanced accuracy and recall, we tuned our GTB parameters as such:

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=5, loss='deviance', max_depth=3,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=10,
              n_iter_no_change=None, presort='auto', random_state=None,
              subsample=1.0, tol=0.0001, validation_fraction=0.1,
              verbose=0, warm_start=False)
```

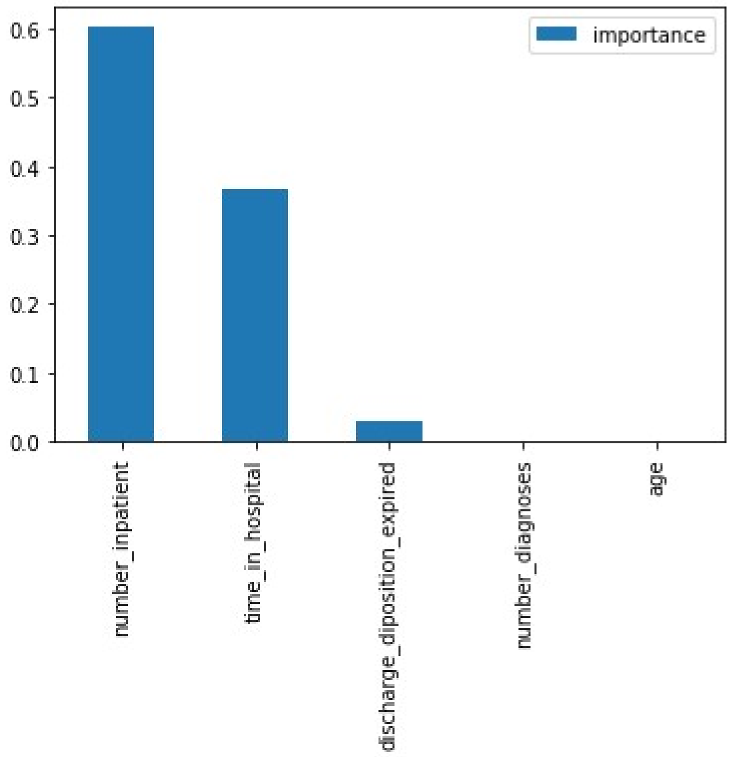
A baseline 10-Fold run gives us:



	train	test	class_0_test	class_1_test
accuracy	0.487737	0.487736	0.459344	0.713669
recall	0.713744	0.713669	0.459344	0.713669
precision	0.142245	0.142232	-	-
f1	0.237215	0.237177	-	-
roc	0.586545	0.586507	-	-

The results are mediocre at best, with a high class 1 recall. While this model would not be a candidate for a final model, we retain it as we may still use it in our ensemble models.

What is interesting to note is resulting feature importance from the GBC model:



Essentially the model only considers important three features, the rest are at zero. The features that actually have non-zero importance values make sense in terms of the context. The variable number_inpatient is referring to inpatient visits, something that can definitely be indicative of future admission to the hospital. The variable time_in_hospital being tied with complexity of treatment and recovery phase, while the expired discharge disposition refers to people who have died and thus will certain not be readmitted. As opposed to adaboost, we see less overfitting with GBC.

10. ENSEMBLE LEARNING

10.1. Majority Vote approach

A straightforward approach to ensemble building is doing a majority vote. As discussed in class, the majority vote of several models, helps reduce overall error, as the the wrong classification is “weeded out”. We wanted to utilize this approach and hopefully improve upon our accuracy and recall for the both classes with such an approach. We also were confident that ensemble models which utilize the predictive power of several models are in general more robust and reliable, given that the diversification of the models can help reduce individual model bias.

In sklearn, there are two types of majority vote: ‘hard’ and ‘soft’. The hard majority vote is essentially just what majority vote stands for, choose the classification that has the most done by the models. The soft majority vote utilizes the predicted probabilities and returns the resulting classification from the sum of the models’ predicted probabilities; in essence, a “blended” prediction probability. In testing our candidate ensemble models, we used both hard and soft voting as the two approaches gave us slightly differing results.

10.2. Logistic Regression approach

Another approach that we used was logistic regression to “predict” the overall prediction. We knew of cases where researchers have applied regression models to the outputs of the models in their ensemble. While it was unknown to us the effect of this approach, we test this ensemble approach and used a logistic model, given our binary output, to fit the predictions of the models and used that final logistic model to predict an overall classification. We found out that utilizing this approach did not bring about significant difference either way in terms of performance, as you can see in the results sheet where we have put the results of the various runs. In testing our final candidate models, we found we had better results with the hard majority vote and sometimes the soft majority vote, so we decided to discontinue this approach in our testing.

10.3. Logistic Regression with probability threshold approach

From the onset, we struggled with developing models that would provide balanced predictions for both classes. One approach that we developed was to tune the thresholds at which classifications were being made. In short, we know that one method that is used to enhance models that are predicting on imbalanced data sets, is providing a specified set of weights or penalty, so that the model won’t treat both classes equally, and thus “pay equal” attention to each classes. The probability threshold however has to do with when a given model produces a set of probabilities for both classes and based on those classes it makes a prediction. In our function, which was taking in a set of different models, thresholds could be specified for each model.

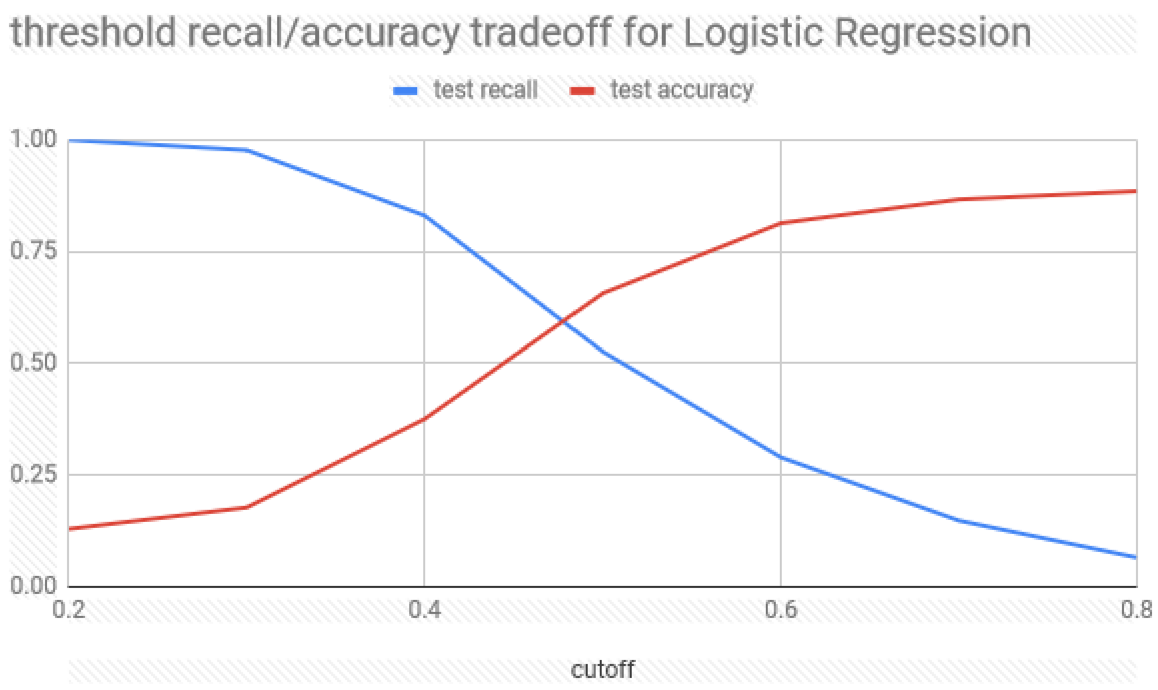
For example, if we had logistic regression as one of the models, we would give it a threshold, and it would make a prediction for class 1 if the given probability of the given instance was above that threshold. As you can see, increasing that threshold would make probability 1 classifying more "strict" and vice versa. Given we had models that gave us high recall, but low accuracy, also models that gave high accuracy, we though this approach could remedy that issue and bring about a ensemble of models with their respective thresholds yielding a high recall, high accuracy model, able to increase signal for class 1.

We noticed that indeed when you tuned these thresholds, they would change the recall and accuracy numbers accordingly, so we knew it was having the effect we wanted. To note, we would then use logistic regression to predict the final prediction from the predictions of the models.

We did extensive runs and testing with this method. While, we did see some increased performance, unfortunately, we did not get a high accuracy/high class 1 recall metric as we desired. While we don't want to show all the runs, it is good to see the model with this approach which got us the best results.

The model was an ensemble of logistic regression, gaussian NB model, multinomial NB model, and random forest, with the respective thresholds of: [0.6,0.5,0.3,0.4]. This attained a train/test accuracy of .811/.704 and a train/test recall of .892/.465. The recall was too low for us to consider this as a final model, but it was an improvement upon the .30 test recall we were getting with previous models. During testing the way test and recall move opposite to each other was evident as we changed around the thresholds.

In order to determine optimal threshold points, we ran different cutoff points to see test recall and test accuracy for each of the models. For example, logistic regression had the following metrics for its probability cutoffs:



It is easy to see that tradeoff occurring as the cutoff value was changed, which is an expected tradeoff of course.

The line in the code where this threshold is applied:

```
(model.predict_proba(test_X)[:,1] > cutoff).astype(int)
```

In short, while this approach did give a slight improvement to our model performance, it still wasn't certainly enough to give us an adequate model, given our below .50 class 1 performance. We needed to do better than this.

10.4. Ensemble Model Candidates

After we used gridsearch to tune our parameters for balanced accuracies and recalls, we finally began to see our metrics going above .50. Given our models with their optimal thresholds, we were confident to proceed to ensembling. While we had done many iterations of the ensembling set both on different datasets and versions of our models, we were now confident with our models and also dataset to move again to this stage.

As mentioned in data cleaning section, we essentially had two clean versions of the data, one visit based and the other unique patient based, but given better results with our visit based data, we used this data for testing.

Our final candidate models were:

1. Logistic Regression
2. Multinomial Naive Bayes
3. Gaussian Naive Bayes
4. Gradient Tree Boosting
5. Adaboost (Decision Tree)
6. Random Forest
7. Bagging (Random Forest)

While not discussed in depth, we also utilized a Bagging Classifier model based on the Random Forest model we had obtained.

Given 7 models, we wanted to test all possible ensemble combinations from these 7 models. We decided to use at least 3 models for our ensemble and thus to test every possible combination (i.e. every combination of 3, of 4, of 5, etc.). Python gives a simple method to obtain those combinations (itertools.combinations). We didn't include 2 model ensembles after we noticed that 2 model ensembles had mediocre performance.

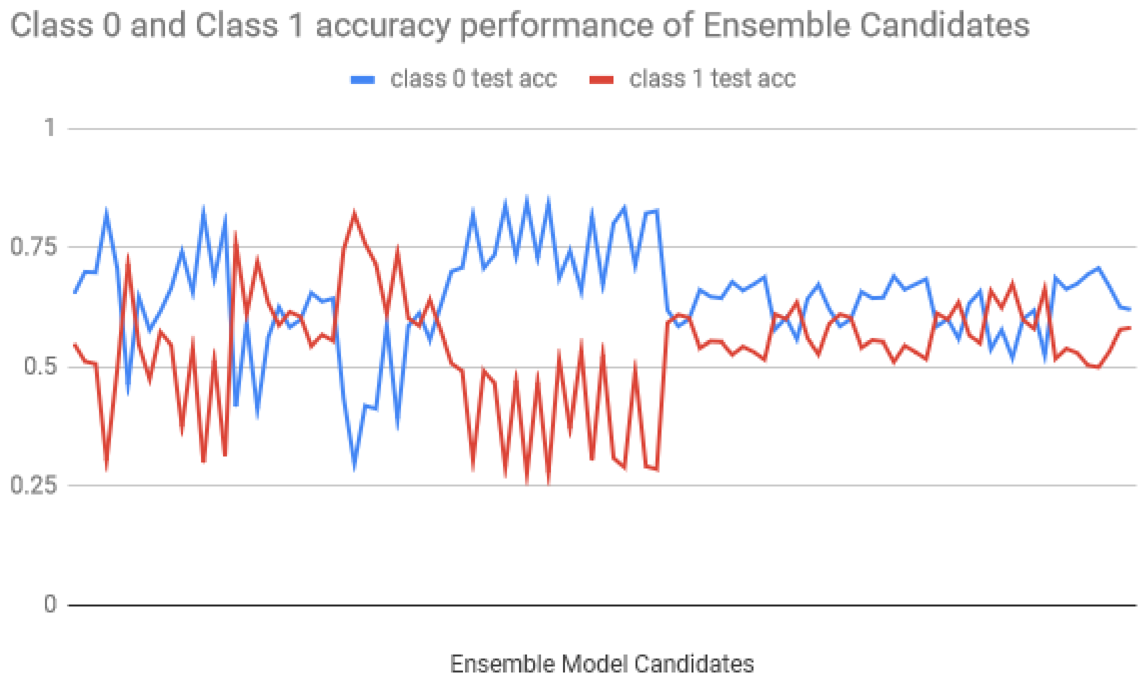
It should be noted that we decided not to include Linear SVM in our final candidate models for ensemble. The main reason is that Linear SVM, given our parameters, constraints, and data structure, had convergence problems. While it was able to output a result, the convergence warning was worrying for us. Even after different methods of scaling to our data, this failure of convergence continued and thus we opted to take this model out of our candidate list. While we had tested its performance in conjunction with other models, the performances were subpar and did not warrant, in our opinion, a deeper investigation.

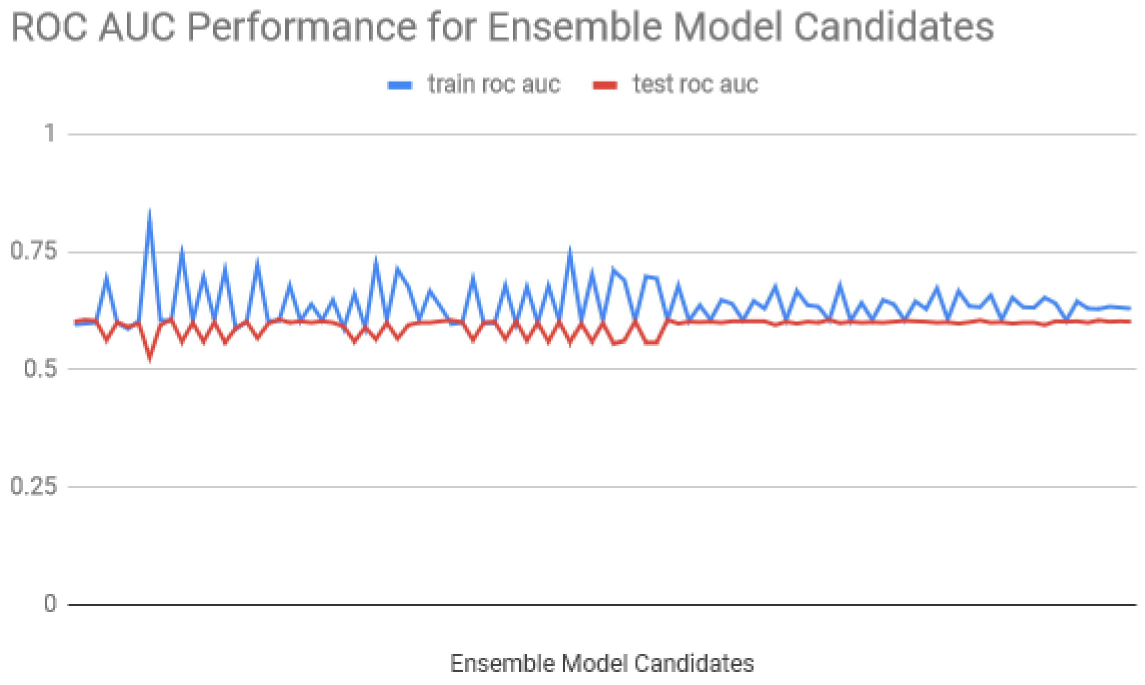
We tested all the combinations on several different settings.

1. Baseline Run - Soft Voting
2. Baseline Run - Hard Voting
3. Random Under Sample Run - Soft Voting
4. Random Under Sample Run - Hard Voting
5. Random Under Sample Run + Min/Max Scalar - Soft Voting
6. Random Under Sample Run + Min/Max Scalar - Hard Voting
7. SMOTE Run - Soft Voting
8. SMOTE Run - Hard Voting

We used all features for our tests, and for the chosen final models, would test with selected features to see either improvement or consistency of the results. For each setting, we ran 99 model combinations. Interestingly, the setting that gave us the best results was Baseline Run with hard voting. While all the settings were similar in terms of performance, the Baseline setting in general, gave us slightly better results. We believe the reason why baseline worked well for us is because in our GridSearch parameter tuning, we were already optimizing for balanced classes and furthermore we were tuning on normalized data. For comparison sake, when we hadn't done GridSearch parameter tuning, our models, in general, performed better with SMOTE and normalization.

While there are a lot of results to show, for convenience sake (you can see the full results in attached data sheets), we show you the performance plots of all the ensemble model candidates with the Baseline Run - Hard Voting setting:





In the first plot, we see the “tradeoff relationship” present with class 0 and class 1 accuracies, while in the second plot we see a consistent gap between test and train area under the curve of the receiver operator curve. From all this data, we were able to hone in on a few candidate ensemble models to choose, based on the parameters were desired for our final model. Choosing 3 models, we were able to run 10-fold tests on them in order to compare and decide which model we would go with.

It should be noted that in general, applying SMOTE, we saw an increase in overfitting, our training metrics increase a bit, but testing metrics falling.

11. RESULTS & FINAL MODEL

11.1. Performance Metrics

We evaluated our models mainly based on two metrics: accuracy and recall rate. In our context, accuracy rate measures the percentage of predictions that correctly predicted whether a patient is readmitted or not (true class 1 and true class 2) among all predictions, and recall measures the percentage of patients that are actually readmitted (correctly predicted true class 1) among all the predictions that a patient would be readmitted (all true class 1).

In this project, it was very easy to optimize for accuracy and get a model with at least 90% accuracy. However, such an accuracy would be misleading and lead to a very poor model performing essentially for mostly one class. Thus, throughout our assessments of models, we paid close attention to the class accuracies and recall because in all it was our goal to produce the most robust and generalizable model possible for both classes. It is for this reason that we opted for models with less accuracy, but importantly more balanced accuracy across our classes. In other words, our most important criteria was to, at a minimum, have a model that would be able to be accurate and recall more than random guessing and for its performance to be balanced both between the classes and also the train/test sets. We believe that a model that has 60% overall accuracy but 60% recall, 60% accuracies between the two classes to be a much more robust model than a model with overall accuracy 92%, but holding a very low recall close to zero.

Another metric we also measured, was the area under the curve of the receiver operator curve (ROC). This metric is often used in data science to assess the performance of a model, as it’s able to discern how well a model is able to produce “signal” thus differentiating the signal from the noise. In a ROC plot, a model that simply randomly guesses will be diagonal straight line, while the more a model is able to produce useful predictions, the more that line will move to the upper left corner (a signal is being read). We wanted chosen models to have roc auc well above random guessing line.

It also should be stated that we stayed true to the Occam's Razor heuristic. For example, if two candidate models perform the same, but one has less features or has less models (if ensemble) then of course we will choose the "simpler" one for consideration of computation time and memory, in parallel, staying true to the heuristic of Occam's Razor.

In short, we want to be clear what criteria we were paying attention to in choosing our models, and this is also apparent in our parameter tuning process. A "good" model is always dependent on the context and data of course. We believe given this highly imbalanced dataset that stability and consistency between the classes, giving useful predictions for both, is what will define a robust model.

11.2. Final Candidate Models

From our ensemble model runs, we were left with three candidate models:

Ensemble 1
Baseline, hard majority vote, all features
Multinomial Naive Bayes, Gaussian Naive Bayes, Gradient boosting, and Random forest

Confusion Matrix		Predicted	
		Class 0	**Class 1**
Actual	**Class 0**	54,068	36,341
	Class 1	4,551	6,806

	Train	Test	class_0_test	class_1_test
Accuracy	60.0%	59.8%	59.8%	59.9%
Recall	60.7%	59.9%	59.8%	59.9%
Precision	16.0%	15.8%	-	-
F1	25.3%	25.0%	-	-
ROC	60.3%	59.9%	-	-

Ensemble 2
Baseline, hard majority vote, all features
Multinomial Naive Bayes, Adaboost, and Random forest

Confusion Matrix		Predicted	
		Class 0	Class 1
Actual	Class 0	58,758	31,651
	Class 1	5,124	6,233

	Train	Test	class_0_test	class_1_test
Accuracy	65.8%	63.9%	65.0%	54.9%
Recall	64.1%	54.9%	65.0%	54.9%
Precision	19.2%	16.5%	-	-
F1	29.5%	25.3%	-	-
ROC	65.1%	59.9%	-	-

Ensemble 3
Baseline, soft majority vote, all features
Logistic Regression, Multinomial Naive Bayes, Gaussian Naive Bayes, Bagging, Adaboost, and Random Forest

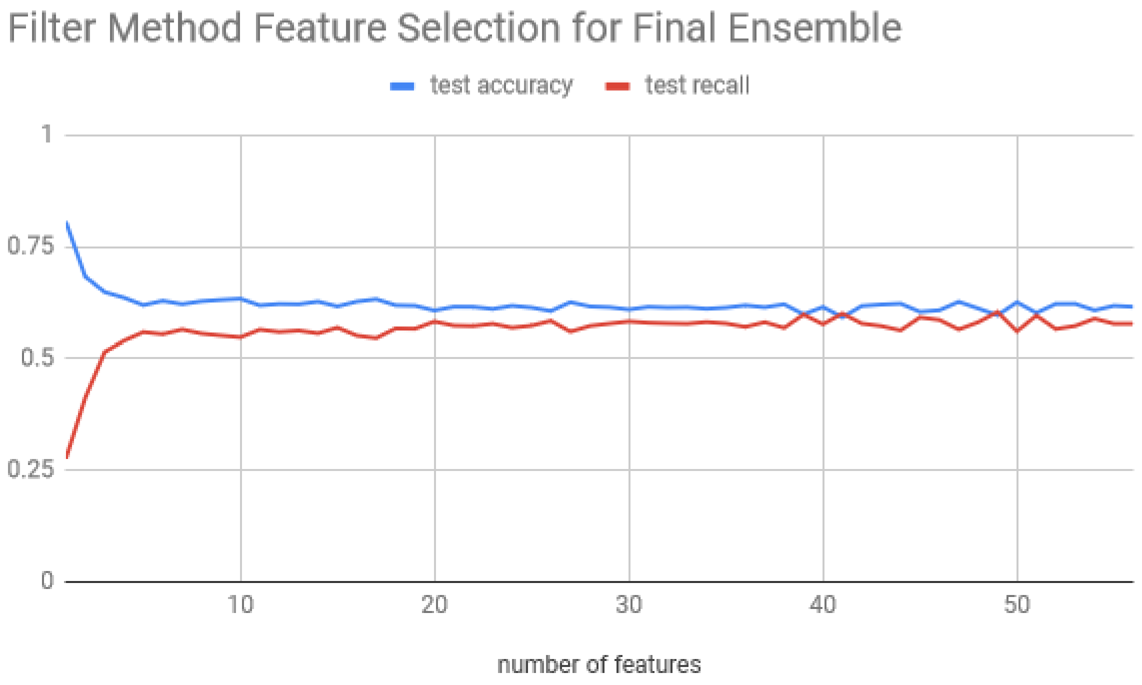
Confusion Matrix		Predicted	
		Class 0	Class 1
Actual	Class 0	54,516	35,893
	Class 1	4,780	6,577

	Train	Test	class_0_test	class_1_test
Accuracy	64.9%	60.0%	60.3%	57.9%
Recall	80.4%	57.9%	60.3%	57.9%
Precision	21.4%	15.5%	-	-
F1	33.8%	24.4%	-	-
ROC	71.7%	59.1%	-	-

11.3. Final Chosen Model

From these three ensembles, we choose ensemble 1 because it is the most balanced out of the three: the criteria that is very important for us. In choosing this model, however, there are a few questions that are raised. 1) Is it valid per statistical assumptions of the models to have both Multinomial and Gaussian Naive Bayes in the same model? 2) Will we get the same performance, if we run the model on the important features determined by random forest?

We first run a feature selectiousing the Filter Method and observe our results:



We notice that approximately after 20 features, our test accuracy and test recall is stabilized, which we are optimizing for. We decide that it is appropriate to use the important features determined by our Random Forest model.

Firstly, to answer the second question, we simply run our ensemble model again, this time with the importance features selected during feature selection by Random Forest. We see the results:

Confusion Matrix		**Predicted**	
		Class 0	**Class 1**
Actual	**Class 0**	53,342	37,067
	Class 1	4,478	6,879

	Train	**Test**	**class_0_test**	**class_1_test**
Accuracy	59.4%	59.2%	59.0%	60.6%
Recall	61.6%	60.6%	59.0%	60.6%
Precision	15.9%	15.7%	-	-
F1	25.3%	24.9%	-	-
ROC	60.3%	59.8%	-	-

While we see a few of the accuracies very slightly decreased, we do see recall slightly increased. Given the very minor difference, but the added benefit of having a model work with less features, we decided to make this decision and keep only the important features. Thus the features we are using for our final model are (38 total features):

number_emergency	time_in_hospital	num_medicine_ change	number_diagnoses
insulin	discharge_diposition_ expired	num_total_medicine	A1Cresult
metformin	number_outpatient	age	diabetesMed
num_lab_procedures	Admission_source_ emergency_room	num_procedures	diag1_circulatory
num_medications	admission_emergency _urgent	diag1_symptoms	diag1_endocrine
is_male	discharge_diposition_ home_other_facility	admission_elective	rosiglitazone
change_in_medications	race_aa	glyburide	admission_source _transfer_hospital _health_care_ facility_clinic
diag1_musculoskeletal	admission_source_ physician_referral	race_white	diag1_injury_ poisoning
admission_unknown	glimepiride	glipizide	pioglitazone
diag1_other	diag1_respiratory		

We believe for the first question, we may be better off indeed without Gaussian Naive Bayes. Firstly, the Gaussian Naive Bayes model is geared for continuous features (something rather lacking in our coded dataset which is mostly categorically coded features) and while some of our features can be seen to come from gaussian distribution, it cannot be said for all. Furthermore, a multinomial NB model is better geared for our mostly discrete features. We test the ensemble without the Gaussian Naive Bayes (keeping only the important features):

Confusion Matrix		**Predicted**	
		Class 0	**Class 1**
Actual	**Class 0**	53,327	37,082
	Class 1	4,476	6,881

	Train	**Test**	**class_0_test**	**class_1_test**
Accuracy	59.4%	59.2%	59.0%	60.6%
Recall	61.6%	60.6%	59.0%	60.6%
Precision	15.9%	15.7%	-	-
F1	25.3%	24.9%	-	-
ROC	60.3%	59.8%	-	-

We see we are getting the same exact result! We for sure take out the Gaussian Naive Bayes out of our ensemble model. Furthermore, we test by removing the other models and notice that our performance significantly goes down if remove. We thus believe we have reached the right balance of models along with a set of the right features to use.

Thus our final ensemble model contains: Multinomial Naive Bayes, Gradient Boosting, and Random Forest with hard majority voting and no normalization or sampling applied to the data, but only the features noted above used.

12. LESSONS LEARNED

Going through this project was definitely a learning experience in many respects. Many of the issues we faced and how we dealt with them, taught us a lot about the challenges that arise in such real world datasets. Firstly, we saw first hand the importance of careful and consistent data cleaning and feature engineering. Not going through that process carefully, can further on significantly affect model performance. In parallel, we realized the importance of assessing the assumptions of statistical independence, feature importance, predictive power of features, and collinearity of features as well. As we went through the steps, especially in the beginning, we found ourselves thinking about these issues, because we knew that later on we would have to deal with them. However, it was so important to think of it in the context of the problem and the medical field.

The beginning, our approach to parameter tuning and modelling was rather naive. While we knew there an imbalance dataset issue, we didn't take the necessary steps to develop our models with a given optimization. So in essence, we were by default optimizing for model accuracy, which in our context was a very wrong approach to take. After extensive tests and modelling, we noticed that while we were able to obtain decent accuracies, our recall scores and class one metrics were subpar (random guessing was doing better). After something thinking and research, we thought that maybe our dataset was cleaned and feature engineered in the wrong way. Thus we went through several iterations of cleaning our data, with different codings, number of features to use, and so forth, but of course it did not improve our performance.

It is only then that we began to focus on tuning our parameters properly using GridSearch and more specifically, optimizing for balanced accuracies and recall. Only after such tuning, we noticed much more balanced results from our models, though the overall accuracies were much less than before, but we were okay with this tradeoff, because for us to have a robust model meant to be performing at a consistent and stable level between class 0 and class 1. Our hope was that with ensemble modelling, we would be able to increase the metrics at this stable level. Unfortunately, ensemble modelling only brought very modest performance gain.

Throughout this process, we rigorously tested models with varying settings, sampling methods, normalizations, and datasets. For datasets, we had two different approaches instance based and patient based. Furthermore, we had to carefully think how applicable certain models would be for our data.

We are sure that it would be possible to obtain a better model than ours, but for us the most valuable lesson from this experience was the process of going through all these steps, failing at times and trying to learn from our failures. Furthermore, applying what we have learned in class, the concepts we had learned really helped us in this endeavour.

If we were to do such a project again, our approach and methods would be much more different and methodical. Given this experience, we would from the onset be much more careful in how we optimized our models from the start. We would take a much more organized approach in terms of our model testing and development. This would save us a lot more time and give us a more "honed in" approach. Furthermore, we believe that the data cleaning not only needs to be done carefully, but there can be different approaches to how the data is cleaned and engineered. With that in mind, what you end up with are many parameters to test on. GridSearch of course simplifies the testing process a lot. In all this experience, gave us a solid framework for how to approach future such data analysis problems.

13. FUTURE WORK

For future work, we would of course try to improve upon our performance while keeping a balanced model. For this, we would try more “exotic” methods for firstly cleaning our data and also in our parameter tuning step. For instance, it would be worthwhile to more carefully research the drugs that are in the data and determine for what reason they are taken and what effects they have. Doing so would give us a better idea about feature engineering them.

Furthermore, it would be worthwhile to research similar readmission data for other diseases and in other locations. Such a diversification of data would provide an opportunity to learn more about the intricacies of our dataset and also what common trends there are independent of disease and location.

Finally, it would be interesting to predict on such data for other target variables. For example, if the patient’s average glucose level has increased or decreased a certain time later. Such a model could show what type of conditions and treatments contribute towards decreasing the effects of diabetes. As stated in the introduction, the medical sphere has a large volume of data and the application of data science in the medical field is becoming more and more pertinent. Where data science can work alongside doctors to provide data driven diagnoses and treatment plans. The future work should try to take this path because one of the greatest achievements of data science can be making people around the world healthier and in turn saving lives as a result of data driven decisions in the medical world.

REFERENCE

Impact of HbA1c Measurement on Hospital Readmission Rates: Analysis of 70,000 Clinical Database Patient Records (https://www.hindawi.com/journals/bmri/2014/781670/)

ICD-9 Codes: https://icd.codes/icd9cm

https://www.datacamp.com/community/tutorials/categorical-data

https://medium.com/@elutins/grid-searching-in-machine-learning-quick-explanation-and-python-implementation-550552200596

https://bigdata-madesimple.com/k-nearest-neighbors-curse-dimensionality-python-scikit-learn/

https://bigdata-madesimple.com/dealing-with-unbalanced-class-svm-random-forest-and-decision-tree-in-python/

APPENDIX - A: ENCODING RULES

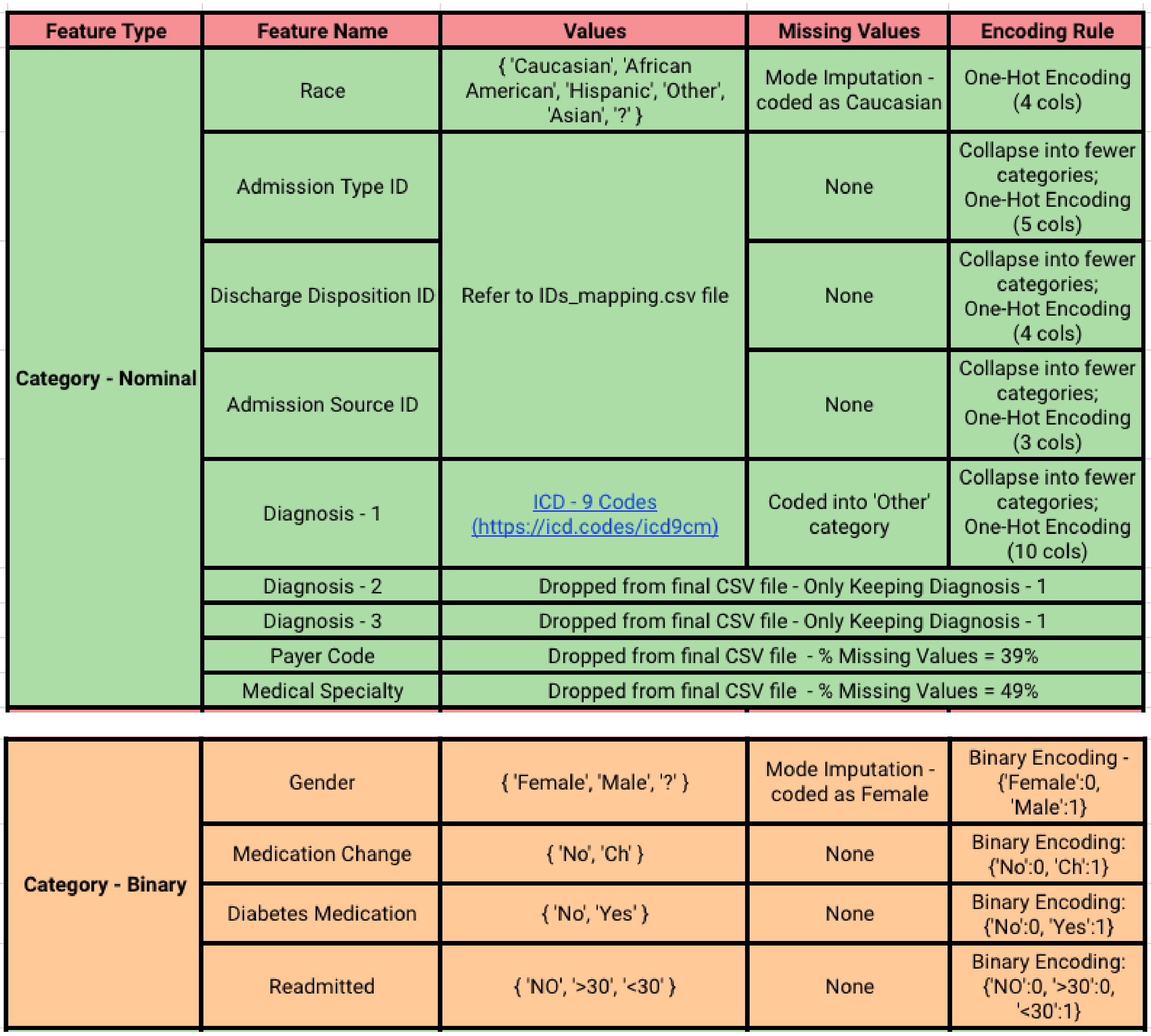
Removed *encounter_id*, *patient_nbr* - unique index values

Removed *weight*, *payer_code*, and *medical_speciality* features - large amount of missing values

Removed *diag_2, diag_3* - focusing on only diag_1 since it has least amount of missing values

Removed ‘*examide*’ and ‘*citoglipton*’ medications - have value “No” for all row instances

Detailed Encoding Rules Table:



Feature Type	Feature Name	Values	Missing Values	Encoding Rule
Category - Nominal	Race	{ 'Caucasian', 'African American', 'Hispanic', 'Other', 'Asian', '?' }	Mode Imputation - coded as Caucasian	One-Hot Encoding (4 cols)
	Admission Type ID	Refer to IDs_mapping.csv file	None	Collapse into fewer categories; One-Hot Encoding (5 cols)
	Discharge Disposition ID	Refer to IDs_mapping.csv file	None	Collapse into fewer categories; One-Hot Encoding (4 cols)
	Admission Source ID	Refer to IDs_mapping.csv file	None	Collapse into fewer categories; One-Hot Encoding (3 cols)
	Diagnosis - 1	ICD - 9 Codes (https://icd.codes/icd9cm)	Coded into 'Other' category	Collapse into fewer categories; One-Hot Encoding (10 cols)
	Diagnosis - 2	Dropped from final CSV file - Only Keeping Diagnosis - 1		
	Diagnosis - 3	Dropped from final CSV file - Only Keeping Diagnosis - 1		
	Payer Code	Dropped from final CSV file - % Missing Values = 39%		
	Medical Specialty	Dropped from final CSV file - % Missing Values = 49%		
Category - Binary	Gender	{ 'Female', 'Male', '?' }	Mode Imputation - coded as Female	Binary Encoding - {'Female':0, 'Male':1}
	Medication Change	{ 'No', 'Ch' }	None	Binary Encoding: {'No':0, 'Ch':1}
	Diabetes Medication	{ 'No', 'Yes' }	None	Binary Encoding: {'No':0, 'Yes':1}
	Readmitted	{ 'NO', '>30', '<30' }	None	Binary Encoding: {'NO':0, '>30':0, '<30':1}

Feature Type	Feature Name	Values	Missing Values	Encoding Rule
Category - Ordinal	Age	Range: '[0, 10)', . . . , '[90, 100)'	None	Binning into 3 Bins & Ordinal Encoding: '**Youth**' = { '[0, 10)', '[10, 20)' }, '**Adult**' = { '[20, 30)', '[30, 40)', '[40, 50)' }, '**Elderly**' = { '[50, 60)', '[60, 70)', '[70, 80)', '[80, 90)', '[90, 100)'} **Coded as (0, 1, 2)**
	Max Glucose Serum	{ 'None', 'Norm', '>200', '>300' }	None	Ordinal Encoding: { 'None':0, 'Norm':1, '>200':2, '>300':3 }
	A1C Results	{ 'None', 'Norm', '>7', '>8' }	None	Ordinal Encoding: { 'None':0, 'Norm':1, '>7':2, '>8':3 }
	21 Medications	{ 'No', 'Steady', 'Up', 'Down' }	None	Ordinal Encoding: { 'No':0, 'Steady':1, 'Up':2, 'Down':2 }
Numerical	No. of Lab Procedures	Discrete value between [1, 132]	None	Binning into 3 Bins & Ordinal Encoding: <=30 : 0, > 30 & <=60 : 1, > 60 : 2
	No. of Procedures	{ 0, 1, 2, 3, 4, 5, 6 }	None	Binning into 3 Bins & Ordinal Encoding: <=2 : 0, >2 & <=4 : 1, > 4 : 2
	No. of Medications	Discrete value between [1, 81]	None	Binning into 3 Bins & Ordinal Encoding: <=20 : 0, > 20 & <=30 : 1, > 30 : 2
	No. of Outpatient Visits	Discrete value between [0, 42]	None	Binary Encoding: {0:0, >0:1}
	No. of Emergency Visits	Discrete value between [0, 76]	None	Binary Encoding: {0:0, >0:1}
	No. of Inpatient Visits	Discrete value between [0, 21]	None	Binning into 3 Bins & Ordinal Encoding: {0:0, 1:1, >1:2}
	No. of Diagnoses	Discrete value between [1, 16]	None	Binning into 4 Bins & Ordinal Encoding: <=4 : 0, >2 & <=6 : 1, >6 & <=8 : 2, > 8 : 3
	Time in Hospital	{ 1, 2, 3, 4, 5, 6, . . ., 14 }	None	Binning into 3 Bins & Ordinal Encoding: <=4 : 0, >4 & <=9 : 1, > 9 : 2
	Weight	Dropped from final CSV file - % Missing Values = 97%		