Predicting Diabetes Hospital Readmission

Udacity Machine Learning Engineer Nanodegree Capstone | 1.8.22

Definition

Project Overview

The goal of this capstone project is to build a model to predict hospital readmissions within a population of patients with diabetes. The dataset used was downloaded from the UCI Machine Learning Repository and sourced from the Health Facts database (Cerner Corporation, Kansas City, MO) as part of a clinical analysis published in BioMed Research International (1). It contains 10 years of diabetic inpatient encounters with 50 corresponding features of patient and clinical attributes. Also included is a “readmitted” target variable indicating whether the encounter resulted in a readmission in <30 days, >30 days, or no readmission. See link to raw data below:
https://archive.ics.uci.edu/ml/datasets/diabetes+130-us+hospitals+for+years+1999-2008

Why predict readmission risk? Readmission risk prediction is a promising machine learning use case with potential to improve both patient outcomes and reduce health care costs. Typically, it is the responsibility of a medical provider to determine when a patient is ready to be discharged from an inpatient facility. This is a crucial decision because discharging a patient too early is likely to lead to poorer health outcomes. Conversely, failing to discharge a patient in a timely manner increases cost unnecessarily and puts pressure on an already overburdened healthcare system. A high performance readmission model could serve as a practical consultative tool to augment a provider's medical expertise when evaluating patients for discharge.

Problem Statement

Using the dataset above described, we will build a training set with the following structure:

- Each row represents an inpatient encounter
- Each column represents a numeric feature (continuous or binary)
- The target variable (“readmitted”) will have 2 levels indicating whether the encounter led to any readmission (1 = <30 days or >30 days, 0 = no readmission)

We will then explore the data visually and perform some statistical tests to assist in feature selection.

From there we will fit two classification algorithms (LinearLearner and XGBoost). Using hyperparameter tuning we will attempt to optimize the performance for each. Finally we will use batch transform to perform inference on the test set and compare the results across models.

Metrics
Models will be evaluated on the following criteria:

- Accuracy (ACC)
- Area under ROC (AUC)

Area under ROC (AUC) curve is a useful evaluation metric because it describes a classifier's ability to rank positive class instances ahead of negative class instances based on their predicted probabilities. In this way, AUC evaluates a classifier without the (arbitrary) need to define a classification threshold.

We will also want to look at Accuracy (ACC). Given that we will be down sampling our training set to an equal number of positive/negative target instances, we can get an intuitive measure of model performance by comparing the model ACC to the expected ACC value of 0.50. Additionally, the Sagemaker LinearLearner algorithm cannot use AUC as the objective function when hyperparameter tuning, so we will want to use ACC in this case.

Analysis
Data Exploration
A script to ingest and extract the raw data can be found in **DataPrep.ipynb** within the project repository. Also included is a mapping file that describes the different types of admissions, discharges, and referral sources.

An overview of the data shows features falling into several domains:
- **ID** - ID fields for patient and event type
- **Demographic** - Race, gender, age, weight
- **Diagnosis** - Diagnosis codes in ICD-9 format
- **Numeric** - Various frequency and time fields (i.e. time in hospital, # meds administered)
- **Lab test** - Categorical results for A1C and Glucose Serum
- **Medication** - Directional dosages for specific medications (up, down, steady, none)
- **Other** - Payer codes (insurance company) and medical specialty of the physician.

When framing the problem we will want to filter the dataset to exclude any encounter that resulted in a death as denoted by the column discharge_disposition_id.

```
## Filter out instances where patient was deceased at discharge ##
df = df[~df.discharge_disposition_id.isin([11,13,14,19,20,21])]
```

```
## View encounter per member ##
df.encounter_id.nunique() / df.patient_nbr.nunique()
1.4193884840691526
```

The result includes data for 99,343 encounters for 69,990 patients (1.42 encounters/patient).

The race and weight demographic fields contain a significant amount of null values. However rather than dropping these records, we will simply encode the null values as dummy encoded categories (1/0). From there we will convert the target column "readmitted":

```
## Convert target variable (reamitted) to binary 1/0 for any readmission type ##
df.readmitted = df.readmitted.apply(lambda x: 0 if x=='NO' else 1)
```

The race and weight demographic fields contain a significant amount of null values. However rather than dropping these records, we will simply encode the null values as dummy encoded categories (1/0). Looking at the ICD9 diagnosis codes, we can see there are many distinct categorical values. We will want to create dummy variables for these, but will use a statistical test to select only those relevant to classification. The integer type fields contain no null values. The lab test and medication fields contain 4 categorical levels. Again these will be dummy encoded and selected based on statistical significance.

Looking at the readmission target variable, we can see there is a slight class imbalance with 47% of encounters resulting in a readmission. We will address this class imbalance in our training dataset through sampling.

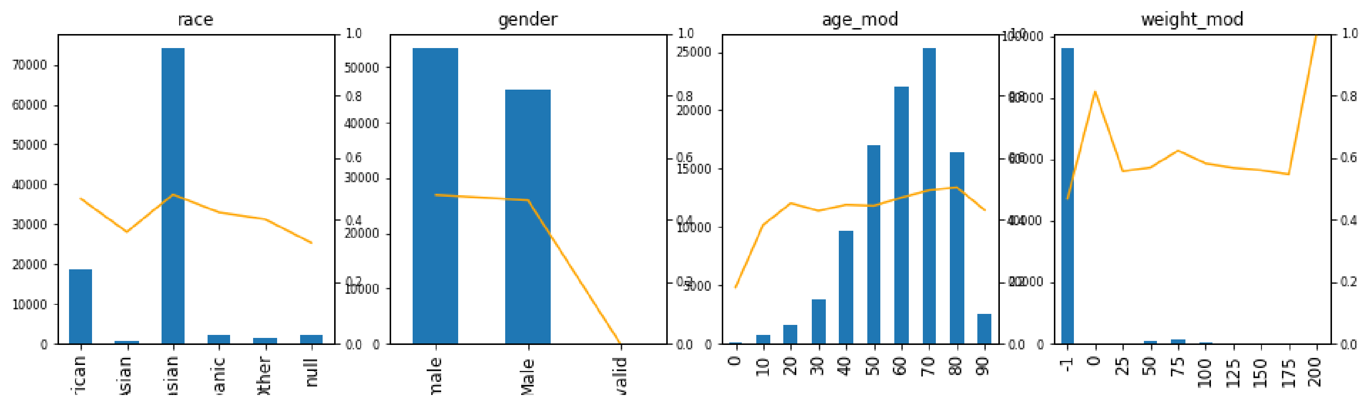
```
## View distribution of target variable (readmitted)
df.readmitted.value_counts()
```

```
0    52527
1    46816
Name: readmitted, dtype: int64
```

Exploratory Visualization

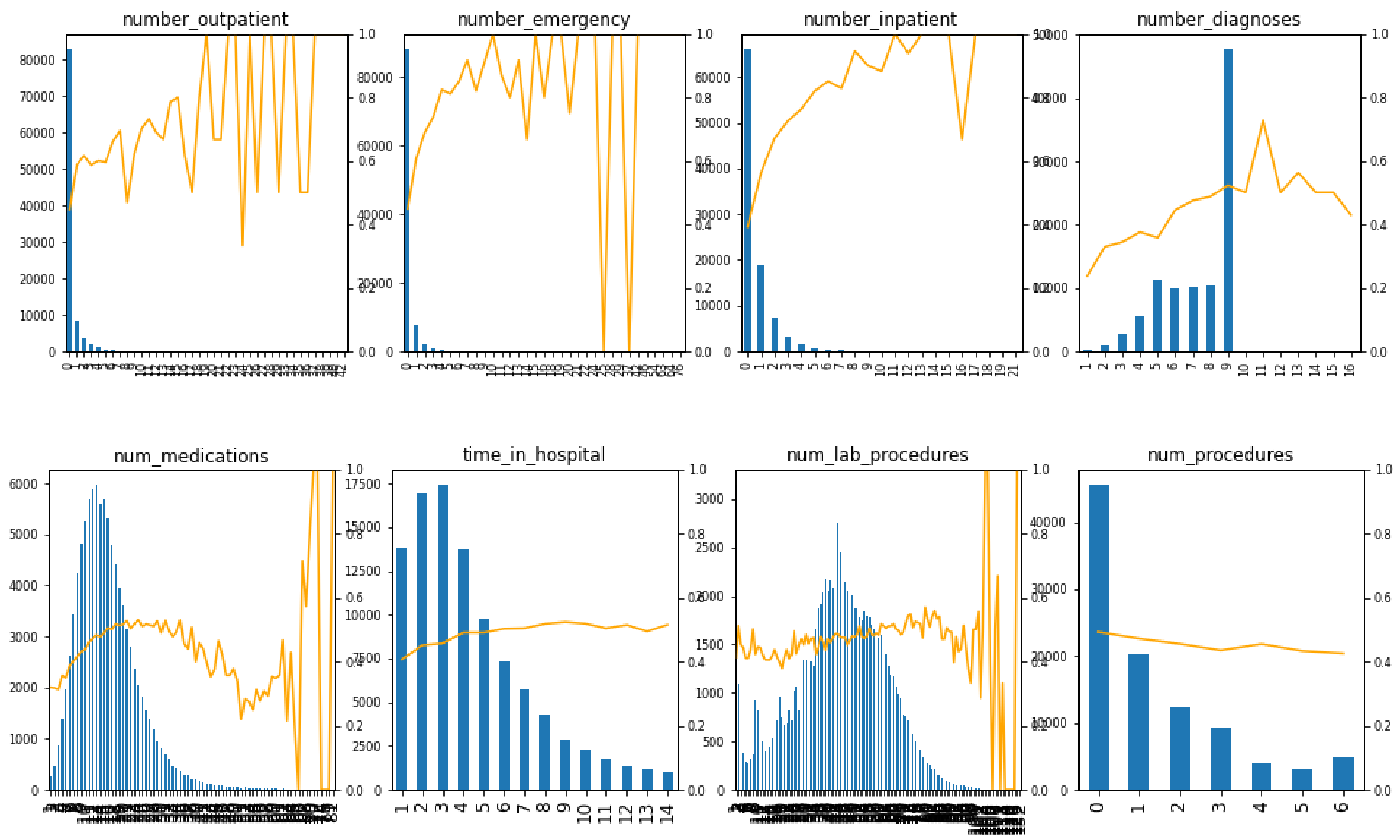
When exploring data it can be informative to create a dual axis view to visualize the prevalence of the target variable (readmissions) across each feature level. For example the below charts show the 4 patient variables (race, gender, age, weight) by:

BARS - # of Encounters
LINE - Readmission Prevalence



Interestingly we see there is not too much variance in readmission prevalence for race and gender for levels with a substantial number of encounters (higher blue bars). As noted, before weight is mostly null (-1). However, we do see a positive correlation between age bucket and readmission prevalence. This is in line with intuition that older patients would be more likely to be readmitted.

We can produce the same view for our numeric features:



Here we see that several of the continuous features show some positive correlation with readmission prevalence. Given that higher values on these fields suggest a more serious inpatient encounter this again aligns with our general intuition on readmission risk.

Algorithms and Techniques

Our approach to predicting readmission events will be to build a binary classifier. As a first pass we will compare two distinct algorithms: Sagemaker LinearLearner and Sagemaker XGBoost.

Classifiers:

Sagemaker LinearLearner (built- in)

As the name suggests, the Sagemaker built-in LinearLearner algorithm fits a linear function to the training data. This function can then be applied to regression, binary classification, or multi-class classification tasks. Our use case will leverage binary classification which is specified as a hyperparameter on the LinearLeaner estimator object.

Sagemaker XGBoost (built-in)
Sagemaker's built-in XGBoost is a non-linear tree based algorithm. This is beneficial in that it allows for modeling more complex relationships within our training data. However, this makes the model more prone to overfitting so this should be taken into account when training/tuning the estimator.

Hyperparameter Tuning
As part of the training process we will perform hyperparameter tuning on both the LinearLearner and XGBoost models. For XGBoost we will optimize on Area under the ROC Curve (AUC). For LinearLearner, Sagemaker doesn't support AUC tuning at present. In this case we will optimize for Accuracy (ACC).

Benchmarking
As a general comparison, I reviewed the meta-analysis Risk Prediction Models for Hospital Readmission: A Systematic Review (2).

Risk Prediction Models for Hospital Readmission: A Systematic Review
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3603349/

While none of the models reviewed exactly match our specific problem statement pertaining to diabetes patients, the analysis does offer good directional information. Depending on the type of readmission and population, readmission models tend to perform somewhere in the AUC (C-statistic) range of 0.60 to 0.70.

Methodology
Data Preprocessing
The first step to implement the above described classification task is to preprocess our data into training and testing datasets. The steps will be as follows:

1. Create dummy fields (binary flags) for all categorical features for all levels
2. Segment target variable y (readmissions) from predictor variables X (features)
3. Perform initial feature selection using a variance threshold approach
4. Perform additional feature selection using a chi square test
5. On our select features, down sample non-readmission (target = 0) encounters to equal the number of readmission encounters (target=1)
6. Implement a test/train/val split to create data subsets for training, validation, and testing.

1. Create dummy fields (binary flags)
```
## Create dummy fields for all categorical variables ##
dummy_fields = fields['DEMO']+fields['DIAG']+fields['OTHER']+fields['LAB']+fields['MED']
dummies = pd.get_dummies(df[dummy_fields])
## Segment numeric fields ##
```

```
numeric_fields = fields['INT']
```

2. Segment target variable y (readmissions) from predictor variables X (features)

```
## Define feature array and target variable ##
X = pd.concat([df[numeric_fields] , dummies], axis=1)
y = df[fields['TARGET']]
```

3. Perform initial feature selection using a variance threshold approach
One approach to feature selection is to remove features with low (or 0 variance). The idea here is that if there is not sufficient variance, then the feature will not be a strong predictor. For example, if a feature has the same value for all records it clearly will not be useful in the prediction task. In our case, we have a sparse dataset with many feature levels encoded as dummy variables. It is likely that many of these features have zero or near zero variance and thus are not relevant to the prediction task.

As a first pass we will define a variance filter function and set the threshold to 0.01.

```
def filter_on_variance(X, threshold = 0.01):

    X_sparse = scipy.sparse.csc_matrix(X.values)
    var_filter = VarianceThreshold(threshold=threshold)
    X_filtered = var_filter.fit_transform(X_sparse)
    filtered_cols = [i for i in X.columns[var_filter.get_support()]]
    X_filtered = pd.DataFrame(X_filtered.todense(), columns = filtered_cols)

    return X_filtered

print ('Feature Selection - Variance Threshold')
print ('Input dimensions: {}'.format(X.shape))
X = filter_on_variance(X, threshold = 0.01)
print ('Output dimensions: {}'.format(X.shape))
```

```
Feature Selection - Variance Threshold
Input dimensions: (99343, 2462)
Output dimensions: (99343, 143)
```

We can see that this has greatly reduced the number of features in our training set from 2,462 to 143.

4. Perform additional feature selection using a chi square test
Another feature selection approach is to compare the relationship between the features and target variable. For example we can implement a Chi-Square test to measure the dependence between each feature and the target (readmissions). The below implementation will filter the features based on a standard statistical significance threshold of p value <= 0.05. Alternatively,

we could also filter on the Chi-Square statistics itself to only include features with a higher level of dependence with the target variable.

```
def chi_filter(X, p_thres = 0.0, chi_thres = 0):
    chi_vals, p_vals = chi2(X, y)
    chi_filter = [True if c>=chi_thres else False for c in chi_vals]
    p_filter = [True if p<=p_thres else False for p in p_vals]
    stat_filter = np.logical_and(chi_filter, p_filter)
    X_filtered = X.loc[:,stat_filter]
    return X_filtered

print ('Feature Selection - Chi Square')
print ('Input dimensions: {}'.format(X.shape))
X = chi_filter(X, p_thres = 0.05, chi_thres = 0.0)
print ('Output dimensions: {}'.format(X.shape))
```

```
Feature Selection - Chi Square
Input dimensions: (99343, 143)
Output dimensions: (99343, 105)
```

By just filtering for "significance" we have further reduced our feature count from 143 to 105.

There are many approaches to feature engineering and this process is one that requires iteration to test different thresholds and selection techniques for optimal model performance. For now our set of 105 features is nice and manageable for a first pass at building our models.

5. Downsample dataset to have equal number of target level instances
As shown before, our dataset has a slight target class imbalance. We can take care of this using a simple function to downsample the majority class to match the number of instances in the minority class.

```
## Down sample majority class (not readmitted) to match the number of readmitted encounters
def downsample(df):
    df_maj = df[df['readmitted']==0]
    df_min = df[df['readmitted']==1]
    df_maj_ds = resample(df_maj,
                         replace=True,
                         n_samples=len(df_min),
                         random_state=42)
    df_ds = pd.concat([df_min, df_maj_ds])
    return df_ds

df = downsample(df)
df.readmitted.value_counts()
```

```
1    46816
0    46816
Name: readmitted, dtype: int64
```

6. Implement a test/train/validation split
Below we first divide the dataset into train and test sets. The test set will be used for generating final evaluation metrics (AUC/ACC). The train set gets further divided into train and validation sets. Sagemaker includes functionality to pass data in distinct training and validation channels when training.

```
## Create train/test/validation sets ##
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.33)
```

Refinement

Below are the areas of refinement implemented in order to improve model performance:

- **Feature Selection** - Our feature selection approach is one that required iteration through varying cutoff thresholds for variance and chi-square p-value. The resulting subset of features perform well as described below, but future enhancements could potentially modify these cutoff points. There is also room for more feature engineering, particularly in terms of the ICD9 diagnosis codes. Grouping these features into higher level clinical categories might reduce noise in the dataset and lift model performance.
- **Model Selection** - As shown below, we will fit a linear and non-linear model to the same data, which effectively diversifies the approach. This allows comparison across model types to select the best performer.
- **Model Tuning -** Within each modeling approach, further refinement is achieved in an automated way using hyperparameter tuning (discussed in detail below).

Model Implementation

LinearLearner

With our processed training data divided into test/train/validation sets we are finally ready to build an estimator. Detailed code can be found in **Train_Model_LinLearner.ipynb.** The first step is to upload our test set to s3 where it can be evaluated using batch transform.

```
# Upload test set from local directory to s3 bucket
test_location = session.upload_data(os.path.join(data_dir, 'test.csv'), key_prefix=prefix)
```

Per Sagemaker's documentation, LinearLearner requires training data to be in "recordIO" format. Below we convert training data into recordio and upload to s3. We repeat again for the validation set.

```
# Convert data to recordio format. Upload training data (and validation) to s3 bucket
buf = io.BytesIO()
```

```
smac.write_numpy_to_dense_tensor(buf, X_train.values.astype("float32"),
y_train.values[:,0].astype("float32"))
buf.seek(0)

key = "recordio-pb-data"
boto3.resource("s3").Bucket(bucket).Object(os.path.join(prefix, "train", key)).upload_fileobj(buf)
s3_train_data = f"s3://{bucket}/{prefix}/train/{key}"
print(f"uploaded training data location: {s3_train_data}")
```

We instantiate a LinearLeaner estimator, specifying an s3 output path for the model artifacts.

```
container = get_image_uri(session.boto_region_name, 'linear-learner')
linear = sagemaker.estimator.Estimator(
    container,
    role,
    train_instance_count=1,
    train_instance_type="ml.c4.xlarge",
    output_path='s3://{}/{}/output'.format(session.default_bucket(), prefix),
    sagemaker_session=session)
```

Next we need to set 2 static hyperparameters:

- ***feature_dim*** - Number of input features in the training data
- ***predictor_type*** - Specify if this is a regression or classification (binary) task

```
linear.set_hyperparameters(feature_dim=105, predictor_type="binary_classifier")
```

We now want to set up a tuning job to find the best values for a subset of tunable hyperparameters. Sagemaker makes this easy by fitting multiple models (max_jobs) while varying the parameters values across each fit. We specify objective_metric_name and objective_type to indicate we want to optimize for maximum accuracy. Once the tuner is instantiated we point it to the training and validation channels in s3 to fit the tuner.

```
## Instantiate tuner instance and input tunable hyperparameter ranges ##
from sagemaker.tuner import IntegerParameter, ContinuousParameter, CategoricalParameter,
HyperparameterTuner
lin_hyperparameter_tuner = HyperparameterTuner(estimator = linear,
                                    objective_metric_name =
'validation:binary_classification_accuracy',
                                    objective_type = 'Maximize',
                                    max_jobs = 20,
                                    max_parallel_jobs = 3,
                                    hyperparameter_ranges = {
                                        'wd': ContinuousParameter(0.1, 1.0),
                                        'l1' : ContinuousParameter(0.1, 1.0),
                                        'learning_rate': ContinuousParameter(0.1, 1.0),
                                    'mini_batch_size': IntegerParameter(100, 5000),
                                    'use_bias': CategoricalParameter([True, False])})

lin_hyperparameter_tuner.fit({'train': s3_train_data, 'validation': s3_val_data})
lin_hyperparameter_tuner.wait()
```

With the hyperparameter job complete, we can extract the best model (highest accuracy) from the tuner object and attach it to a new estimator object. With the new estimator we can launch a batch transform job to process our test data and write the result to the model's output path.

```
lin_best = sagemaker.estimator.Estimator.attach(lin_hyperparameter_tuner.best_training_job())
lin_transformer = lin_best.transformer(instance_count = 1, instance_type = 'ml.m4.xlarge')
lin_transformer.transform(test_location, content_type='text/csv', split_type='Line')
lin_transformer.wait()
```

Finally, we can evaluate our test data results to calculate ACC and AUC performance metrics.

```
## Download transform output from s3 location ##
!aws s3 cp --recursive $lin_transformer.output_path $data_dir/linreg_results

## Calculate test accuracy and auc performance ##
results = pd.read_csv(os.path.join(data_dir, 'linreg_results/test.csv.out'), header=None)
predictions = [int(p.split(':')[1]) for p in results[0]]
probs = [float(p.split(':')[1].split('}')[0]) for p in results[1]]

from sklearn.metrics import accuracy_score, roc_auc_score
print ('LinLearner test accuracy: {}'.format(accuracy_score(y_test, predictions)))
print ('LinLearner test auc: {}'.format(roc_auc_score(y_test, probs)))
LinLearner test accuracy: 0.6274960354704036
LinLearner test auc: 0.6743540445370801
```

The LinearLearner model achieved a test ACC of 0.63 and test AUC of 0.67. These results appear reasonable as they are right within our benchmark range of 0.60-0.70. Next we will fit the non-linear XGBoost model and compare the results.

XGBoost

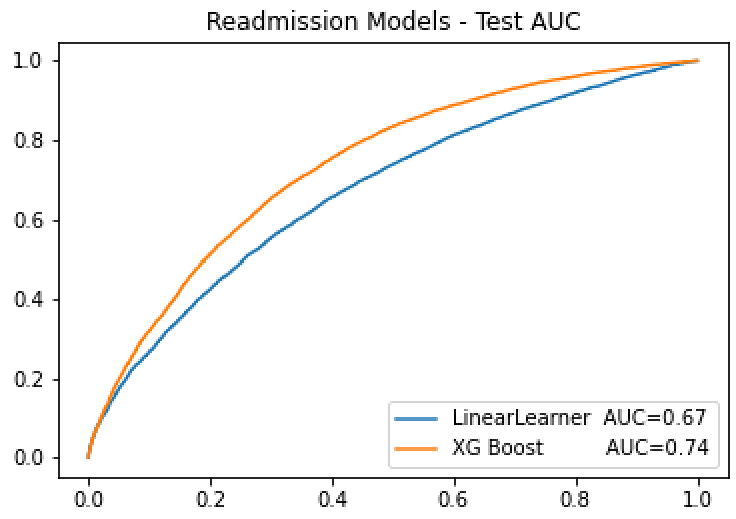
To build an XGBoost model we simply repeat much of the process outlined above. See **Train_Model_XGBoost.ipynb**. One key difference lies in our tuner object where we specify different hyperparameter ranges and objective metric (AUC).

```
xgb_hyperparameter_tuner = HyperparameterTuner(estimator = xgb,
                                    objective_metric_name = 'validation:auc',
                                    objective_type = 'Maximize',
                                    max_jobs = 20,
                                    max_parallel_jobs = 3,
hyperparameter_ranges = {
    'max_depth': IntegerParameter(3, 12),
    'eta'      : ContinuousParameter(0.05, 0.5),
    'min_child_weight': IntegerParameter(2, 8),
    'subsample': ContinuousParameter(0.5, 0.9),
    'gamma': ContinuousParameter(0, 10)})
```

```
## Calculate test accuracy and auc performance ##
output = pd.read_csv(os.path.join(data_dir, 'xgb_results/X_test.csv.out'), header=None)
predictions = [round(num) for num in output.squeeze().values]
probs = [p for p in output.squeeze().values]

from sklearn.metrics import accuracy_score, roc_auc_score
print ('XBG test accuracy: {}'.format(accuracy_score(y_test, predictions)))
print ('XBG test auc: {}'.format(roc_auc_score(y_test, probs)))
XBG test accuracy: 0.6754911162173534
XBG test auc: 0.737488341310636
```

Interestingly, we see substantial test performance improvement using the non-linear XGBoost model. A nice way to compare multiple model performance is to plot both on a common ROC curve:



Results

Comparing final results for our models we can see both had strong performance relative to the general benchmark of 0.60-0.70 AUC. The XGBoost model was substantially more performant and showed markedly stronger predictive power on the test set. However, LinearLearner was not optimized on AUC, so additional manual hyperparameter tuning might provide some marginal performance improvement. Given that the data was partitioned into distinct training/validation/test sets we have reasonable confidence that the model is not overfit. However, the performance to benchmark might be misleading in that we are drawing from a population of only diabetic patients. As this cohort has higher readmission risk compared to the general population, this might be an easier prediction task compared to predicting all cause readmission on the general population.

In a production context we would certainly want to closely monitor the model's performance on real world data. Critical to this process would be analyzing the model parameters to understand which features are the primary drivers of readmission. This is an area for further exploration on this project and could also inform additional feature engineering efforts. At this time it does not appear that Sagemaker offers default functionality to extract and analyze model parameters for built-in algorithms per the below Sagemaker forum post:

- https://forums.aws.amazon.com/thread.jspa?threadID=272345

Developing some sort of custom solution as outlined above will be the focus of additional efforts on this project.

References

1. Beata Strack, Jonathan P. DeShazo, Chris Gennings, Juan L. Olmo, Sebastian Ventura, Krzysztof J. Cios, and John N. Clore, "Impact of HbA1c Measurement on Hospital Readmission Rates: Analysis of 70,000 Clinical Database Patient Records," BioMed Research International, vol. 2014, Article ID 781670, 11 pages, 2014.
2. Kansagara D, Englander H, Salanitro A, et al. Risk prediction models for hospital readmission: a systematic review. *JAMA*. 2011;306(15):1688-1698. doi:10.1001/jama.2011.1515