nallen_HW6

November 15, 2025

1 A6 Clinical Predictive Modeling [ ]

Welcome to A6! Please enter answers to the questions in the specified Markdown cells below, and complete the code snippets in the associated python files and ipynb cells as specified. When you are done with the assignment, follow the instructions at the end of this assignment to submit.

Grading: All answers will be graded on the correctness and quality of your code and analyses. Partial credit will be given based on a demonstration of conceptual understanding and how close you can come to solving the problem. At various points we will ask you to produce particular values: the correctness of these numbers will not be used for your grade - they are tools for us to get an idea about what your code is doing.

1.0.1 Learning Objective

In this assignment, you will work with a dataset very similar to what you analyzed in A5, to train and evaluate a machine learning model that predicts whether a patient will die during their ICU stay.

The first portion of this assignment is identical to the first portion of A5, to create the data used in subsequent sections.

1.0.2 Resources

- Pandas Cheat Sheet : `https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf`

1.0.3 Environment Set-Up

To begin, we will need to set up an virtual environment with the necessary packages. A virtual environment is a self-contained directory that contains a Python interpreter (aka Python installation) and any additional packages/modules that are required for a specific project. It allows you to isolate your project's dependencies from other projects that may have different versions or requirements of the same packages.

In this course, we require that you utilize Miniconda to manage your virtual environments. Miniconda is a lightweight version of Anaconda, a popular Python distribution that comes with many of the packages that are commonly used in data science.

Instructions for setting up your environment using Miniconda:

1. If you do not already have Miniconda installed, download and install the latest version for your opperating system from the following link: https://docs.conda.io/en/latest/miniconda.html#latest-miniconda-installer-links

2. Create a new virtual environment for this assignment by running the following command in your terminal:

   ```
   conda env create -f environment.yml
   ```

   This will create a new virtual environment called `biomedin215`

3. Activate your new virtual environment by running the following command in your terminal:

   ```
   conda activate biomedin215
   ```

   This will activate the virtual environment you created in the previous step.

4. Finally, ensure that your `ipynb` (this notebook)'s kernel is set to utilize the `biomedin215` virtual environment you created in the previous steps. Depending on which IDE you are using to run this notebook, the steps to do this may vary.

[54]:
```
# Run this cell:
# The lines below will instruct jupyter to reload imported modules before
# executing code cells. This enables you to quickly iterate and test revisions
# to your code without having to restart the kernel and reload all of your
# modules each time you make a code change in a separate python file.

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[55]:
```
# Run this cell to ensure the environment is setup properly
import pandas as pd
import numpy as np
import os
import warnings

print("Imported modules successfully.")
```

```
Imported modules successfully.
```

1.0.4 *Note to Students:*

Throughout the assignment, we have provided `sanity checks`: small warnings that will alert you when your implementation is different from the solution. Our goal in providing these numbers is to help you find bugs or errors in your code that may otherwise have gone unnoticed. Please note: the sanity checks are just tools we provided to be helpful, and should not be treated as a target to hit. We manually grade each assignment based on the code you submit, and not based on whether you get the exact same numbers as the sanity checks.

If you are failing the sanity checks (even by a lot) and your implementation is correct with minor errors, you will still receive the majority of the points (if not all the points).

[56]:
```python
# Run this cell to set up sanity checks warnings
# Note: You do not need to change anything in this cell

# Creates a custom warning class for sanity checks
class SanityCheck(Warning):
    pass

# Sets up a cosutom warning formatter
def custom_format_warning(message, category, filename, lineno, line=None):
    if category == SanityCheck:
        # Creates a custom warning with orange text
        return f'\033[38;5;208mSanity Check - Difference Flagged:
 ↪\n{message}\033[0m\n'

    return '{}:{}: {}: {}\n'.format(filename, lineno, category.__name__,␣
 ↪message)

# Sets the warning formatter for the entire notebook
warnings.formatwarning = custom_format_warning
```

1.0.5 Data Description

In this assignment, you will work with a dataset that we have prepared for you using a process similar to what you did in previous homeworks. The dataset describes patients from the MIMIC III database who were put on mechanical ventilation and were stable for 12 hours. Some of these patients then experienced a sudden and sustained drop in oxygenation, while others did not. You will practice using common time-saving tools in the **Pandas** library and **Python** programming language that are ideally suited to these tasks.

We have recorded a variety of features about each patient before the 12-hour mark (the index time), including counts of all prior diagnoses (aggregated with IC), all respiratory-related concepts in their notes, and indicators of events recorded in the patient charts. Indicator features are the number of times each event was recorded in the patient record, regardless of what the measured value was. For those chart events which have numeric values associated with them (e.g. lab tests) we found those in which a value was recorded for over 85% of the cohort and included the latest recorded value of those features. In addition, we have included demographic features (age and sex). For the small number of patients who did not have one or more of those features recorded, we used column-mean imputation to impute them. We also recorded whether or not each patient went on to experience a sudden and sustained drop in their oxygenation (the exposure). Finally, we recorded whether or not each patient eventually died during their hospitalization (the outcome). All of this data is contained in `patient_feature_matrix.csv`.

The companion file `feature_descriptions.csv` has descriptions of each of the features and their provenance.

The final dataset you have access to is called `cohort.csv`, which contains the index time, exposure time (if any), in-hospital time of death (if any), and the time of censoring (when the patient was released from the hospital).

All of the data you need for this assignment is available on Canvas.

Once you have downloaded and unzipped the data, you should see the following `3` csv files: - `patient_feature_matrix.csv`

- `cohort.csv`
- `feature_descriptions.csv`

Specify the location of the folder containing the data in the following cells:

[57]:
```
# Specify the path to the folder containing the data files
data_dir = "/Users/nickallen/Documents/GitHub/BMDS215/A6/data"
```

[58]:
```
# Run this cell to make sure all of the files are in the specified folder
expected_file_list = ["patient_feature_matrix.csv", "cohort.csv",␣
 ↪"feature_descriptions.csv"]

for file in expected_file_list:
    assert os.path.exists(os.path.join(data_dir, file)), "Can't find file {}".
 ↪format(file)

print("Sanity check: Success")
```

```
Sanity check: Success
```

[59]:
```
# Run this cell to load the data from the CSV files into Pandas DataFrames
patient_feature_matrix = pd.read_csv(os.path.join(data_dir,␣
 ↪"patient_feature_matrix.csv"))
cohort = pd.read_csv(os.path.join(data_dir, "cohort.csv"))
feature_descriptions = pd.read_csv(os.path.join(data_dir, "feature_descriptions.
 ↪csv"))
```

2 1 (0 Points) Preprocessing

Lets start by preprocessing the data (just like we did in A5). We have provided you with a file called `preprocess.py` that contains functions that perform the preprocessing steps you implemented in `A5`. You do not need to modify this file or code in the following cells.

2.1 1.1 (0 Points) Create Feature Matrix and Outcome Vector

Run the following cell to split the patient feature matrix up into `features` and `labels` dataframes.

[60]:
```
from src.preprocessing import split_labels_and_features

# Run this cell to split the labels from the features
features, labels = split_labels_and_features(patient_feature_matrix,
                                              {"oxy_drop": {"stable": 0,␣
↪"oxy_drop":1},
                                               "death_in_stay": {"survived": 0,␣
↪"died": 1},
                                               "gender": {"M":0, "F": 1}})
```

`Important note to students:` It's worth noting that the definition of `gender` in the `MIMIC-III` database and many other clinical databases is a simplified representation of gender that does not account for a range of gender identities and expressions, and most often isn't distinguished from biological sex which may or may not reflect an individual's self-identified gender identity (SIGI). Increasingly, EHR schemas are updated/designed to capture SIGI in addition to biological sex to improve care for transgender and gender nonconforming (TGNC) patients.

As you continue your journey in medical data science, remember that it is critically important to always review the schema of the dataset you are working with to ensure that you know how different data fields are defined, so that you can fully understand the assumptions and limitations of your downstream analysis.

2.2 `1.2 (0 Points)` Removing Uninformative Features

Before we do any modeling, let's cut down on our feature space by removing `low-variance features` that probably aren't useful enough to measure association with or use in a predictive model.

Run the following cell.

[61]:
```
from src.preprocessing import feature_variance_threshold

# Run this cell to remove features with low variance
filtered_features = feature_variance_threshold(features, freq_cut=95/5,␣
 ↪unique_cut=0.1)
```

For the rest of the assignment, we will be working with this filtered version of the dataframe.

3 `2 (100 Points)` Predictive Analyses

In this section we will train a predictive model utilizing supervised learning techniques to predict whether a patient will die during their hospitalization, given only the data from before the end of their 12-hour long stable ventilation period (prior to the index time).

3.1 `2.1 (5 Points)` Creating training and test sets

The first step of any predictive modeling task is to split our dataset into training and test sets: subsets of the data that we will use to train and evaluate our model respectively.

The `train` set is a subset of the original data set on which the machine learning model is trained. It consists of input-output pairs where the inputs are the features or predictors, and the outputs are the labels or targets the model aims to predict. During the training phase, the model learns to recognize patterns or mappings from inputs to outputs. The quality and diversity of the training data are crucial, as they determine the model's ability to generalize to new, unseen data.

The `test` set is a **seperate** subset of the original data that is used to evaluate the performance of the model after the training phase. It is crucial that the data in the test set has not been used during training. The purpose of the test set is to assess how well the model generalizes to new data - that is, its predictive performance on examples it hasn't encountered during training.

Utilizing a training/test split is a fundamental concept in machine learning and data science workflows that allows us to get an estimate of how well our model will perform on unseen data, and determine if our model is overfitting to the training data. In real world applications, it is a best practice to split the data into three subsets: `train`, `validation`, and `test` sets. The `validation` set is used to assess the performance of the model on unseen data during the training phase, and is used to tune the hyperparameters of the model. By creating a `validation` set, the `test` set can be reserved to only be utilized to evaluate the final performance of the model after the training phase is complete.

For simplicity, we will only be utilizing `train` and `test` sets in this assignment.

Let's randomly split the examples in our dataset into `train` and `test` sets. We will use 80% of the data for training, and 20% for testing. Implement the function `split_train_test` in `data.py` to split the data into `train` and `test` sets.

[62]:
```
from src.data import split_data

# Run this cell to split the data into training and test sets
train_features, test_features, train_labels, test_labels =␣
↪split_data(filtered_features, labels, test_fraction=0.2, random_state=42)

# Sanity Check
if train_features.shape[0] != train_labels.shape[0]:
    warnings.warn("Training features and labels have different number of␣
↪samples", SanityCheck)
if test_features.shape[0] != test_labels.shape[0]:
    warnings.warn("Test features and labels have different number of samples",␣
↪SanityCheck)

# Ensure all rows in train_features and train_labels have same index
if not train_features.index.equals(train_labels.index):
    warnings.warn("Indices of train_features and train_labels are not equal",␣
↪SanityCheck)
# Ensure all rows in test_features and test_labels have same index
if not test_features.index.equals(test_labels.index):
    warnings.warn("Indices of test_features and test_labels are not equal",␣
↪SanityCheck)

# Reproducibility check: Ensure first index is
if train_features.index[0] != 13010:
    warnings.warn(f"First index is not correct. Don't forget to set your random␣
 ↪state!", SanityCheck)

# Ensure test features is test_fraction of total features
if test_features.shape[0] != int(0.2 * features.shape[0]):
    warnings.warn("Test features do not contain correct number of samples",␣
 ↪SanityCheck)

# Ensure train features is 1-test_fraction of total features
if train_features.shape[0] != int(0.8 * features.shape[0]):
    warnings.warn("Train features do not contain correct number of samples",␣
 ↪SanityCheck)
```

[63]:
```
# Some of the functions we will use later require that labels are 1D arrays␣
 ↪instead of DataFrames.
# Run this cell to convert the labels to 1D arrays
# NOTE: No need to change anything in this cell
train_labels = train_labels.values.ravel() # train_labels is now a 1D numpy␣
 ↪array
test_labels = test_labels.values.ravel()   # test_labels is now a 1D numpy array
```

3.2 2.2 (30 Points) Exploratory Modeling

Let's explore a few different models we can use to perform our task!

3.2.1 2.2.1 (10 Points) Exploratory Logistic Regression

Regularization is a fundamental concept in machine learning and statistics, particularly in the context of regression and classification problems. It involves adding a penalty to the loss function that the algorithm is trying to minimize. The primary goal of regularization is to prevent overfitting, which occurs when a model learns the training data too well, including the noise, and performs poorly on unseen data.

There are many different types of regularization, but the two most common types are `L1` and `L2` regularization. Let's talk a bit more about these:

L1 Regularization Also known as `Lasso` regularization, `L1` regularization adds a penalty term to the loss function that is proportional to the sum of the absolute values of the coefficients of the model:

$ Penalty( ) = *{j=1}^{p}* / {j}|$

In the equation above, β_j are the coefficients of the model, while λ is a hyper-parameter that controls how much the penalty term impacts the total loss. When $\lambda = 0$, the penalty term has no effect, and the lasso model reduces to a standard linear regression model. As λ increases, and the regularization's impact on the total loss becomes greater.

L1 regularization induces `sparsity`, which means that it encourages as many coefficients in the model to be 0 as possible. L1 regularization is often beneficial in contexts where feature selection is important, as it simplifies the model by retaining the parameters that correspond to only the most important features.

L2 Regularization Also known as `Ridge` regularization, `L2` regularization adds a penalty term to the loss function that is proportional to the sum of the squared values of the coefficients of the model:

$ Penalty( ) = *{j=1}^{p}* {j}^2$

Like in the equation for `L1` regularization, β_j are the coefficients of the model, while λ is a hyperparameter that controls how much the penalty term impacts the total loss.

Since L2 regularization penalizes the squared values of the coefficients, it discourages individual coefficients from becoming very large. This results in a model with smaller, more evenly distributed coefficient values. This can help stabilize the estimation, especially in the presence of highly correlated features.

For this first problem, let's attempt to implement a logistic regression model with `L2` regularization. Implement the function `logistic_regression` in `models.py`. When you are finished, run the following cell.

[64]:
```
from src.models import logistic_regression

# Run this cell to fit the model
# NOTE: You may see a warning about failed convergence. This is ok!
log_reg_model = logistic_regression(train_features, train_labels, lambda_=1e-4,␣
 ↪max_iter=10000)

# Sanity Check:
if not hasattr(log_reg_model, "predict"):
    warnings.warn("Ensure the returned object is sklearn.linear_model._logistic.
 ↪LogisticRegression", SanityCheck)
```

```
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
```

Using the model we just trained, we can now make predictions on the test set!

[65]:
```
# Run this cell to make predictions on the test set. No need to change anything␣
 ↪in this cell
test_predictions = log_reg_model.predict(test_features)
```

Let's characterize the performance of our model by calculating the **accuracy**. Implement the function `accuracy` in `metrics.py` and run the following cell to calculate the accuracy of our model.

```
[66]: from src.metrics import accuracy
      # Run this cell to calculate the accuracy of the model.
      log_reg_accuracy = accuracy(test_labels, test_predictions)

      # Display the accuracy of the model
      print(f"Accuracy of the model on the test set: {log_reg_accuracy:.3f}")

      # Sanity Check:
      if log_reg_accuracy < 0.8:
          warnings.warn("Accuracy is too low, something is likely wrong with the␣
       ↪model set up or accuracy calculation", SanityCheck)
```

```
Accuracy of the model on the test set: 0.813
```

If we only look at the accuracy metric we calculated above, our performance looks pretty good! Unfortunately, accuracy can be misleading. To understand why this is the case, let's do a small experiment.

Let's say we had a model that ALWAYS output a 0 for every example. What would the accuracy of this model be? Run the following cell to find out:

```
[67]: # Run this cell! (No need to change anything)

      # This line creates an array of zeros with the same shape as the test labels
      # This will act like the predictions from a model that predicts all zeros
      all_zero_predictions = np.zeros_like(test_labels)

      # Let's determine the accuracy of a model that predicts all zeros
      all_zero_accuracy = accuracy(test_labels, all_zero_predictions)

      # Display the accuracy of the model
      print(f"Accuracy of model that just predicts all zeros on the test set:␣
       ↪{all_zero_accuracy:.3f}\n")

      # Display class balance
      print(f"Class balance:\n{pd.Series(test_labels).value_counts(normalize=True)}")
```

```
Accuracy of model that just predicts all zeros on the test set: 0.806

Class balance:
0    0.806078
1    0.193922
Name: proportion, dtype: float64
```

Accuracy is not always a suitable metric for evaluating the performance of a model on a binary classification problem, especially when there is a significant **class imbalance**, like in this case where there's a majority of the '0' class. In this case, a model that always outputs '0' will have a high accuracy, but is clearly not a good model.

Let's explore some other metrics we can use to evaluate the performance of our model!

3.2.2 `2.2.2 (5 Points)` Confusion Matrix and Performance Metrics

One of the best ways we can evaluate the performance of a binary classification model is by utilizing a **confusion matrix**.

A confusion matrix is a table often used to describe the performance of a classification model on a set of test data for which the true values are known. It allows you to visualize the accuracy of the model by comparing the actual target values with those predicted by the model.

Here is what a typical confusion matrix looks like for a binary classification problem. Each cell in the matrix represents the number of examples that fall into one of the four categories: - True Positive: The model predicted the example to be positive, and the actual label was positive. - False Positive: The model predicted the example to be positive, but the actual label was negative. - True Negative: The model predicted the example to be negative, and the actual label was negative. - False Negative: The model predicted the example to be negative, but the actual label was positive.

Confusion Matrix Example

	Predicted: (0)	Predicted: (1)
Actual Label: (0)	# True Negative (TN)	# False Positive (FP)
Actual Label: (1)	# False Negative (FN)	# True Positive (TP)

Implement the function `confusion_matrix` in `metrics.py` to calculate the confusion matrix for our model. When you are done, run the following cell to calculate the confusion matrix for our model.

[68]:
```python
from src.metrics import confusion_matrix

# Run this cell to calculate the confusion matrix
con_matrix = confusion_matrix(test_labels, test_predictions)

# Display the confusion matrix
print("Confusion Matrix:")
print(con_matrix)

# Sanity Check: Ensure the confusion matrix is the correct shape
if con_matrix.shape != (2, 2):
    warnings.warn("Confusion matrix is not the correct shape", SanityCheck)

# Sanity Check: Ensure total of confusion matrix cells equals number of test␣
 ↪samples
if con_matrix.sum() != test_labels.shape[0]:
    warnings.warn("Confusion matrix does not contain correct number of␣
 ↪samples", SanityCheck)
```

```
Confusion Matrix:
[[528.  29.]
 [100.  34.]]
```

Another nice thing about confusion matrices is that they allow us to easy calculate a variety of other metrics that are useful for evaluating the performance of our model. Let's implement a few of these metrics!

Sensitivity (also called the true positive rate, or recall in some fields) measures the proportion of actual positives that are correctly identified by the test. It answers the question: "Of all the people who actually have the disease, how many did we correctly diagnose?"

> **Recall** is the same as **sensitivity**: it measures the proportion of actual positives that are correctly identified. The term "recall" is more commonly used in machine learning, while "sensitivity" is more commonly used in medicine.

Implement the function `sensitivity` in `metrics.py` to calculate the sensitivity of our model. When you are done, run the following cell to calculate the sensitivity of our model.

[69]:
```python
from src.metrics import sensitivity

# Run this cell to calculate the sensitivity
sensitivity_value = sensitivity(con_matrix)

# Display the sensitivity
print(f"Sensitivity of the model on the test set: {sensitivity_value:.3f}")
```

```
Sensitivity of the model on the test set: 0.254
```

Specificity (also called the true negative rate) measures the proportion of actual negatives that are correctly identified by the test. It answers the question: "Of all the people who do not have the disease, how many did we correctly identify as not having the disease?"

Implement the function `specificity` in `metrics.py` to calculate the specificity of our model. When you are done, run the following cell to calculate the specificity of our model.

[70]:
```python
# Run the cell below to calculate the specificity
from src.metrics import specificity

specificity_value = specificity(con_matrix)

# Display the specificity
print(f"Specificity of the model on the test set: {specificity_value:.3f}")
```

```
Specificity of the model on the test set: 0.948
```

Precision (also called positive predictive value) measures the proportion of positive identifications that were actually correct. It answers the question: "Of all the people we diagnosed with the disease, how many actually have the disease?"

Implement the function `precision` in `metrics.py` to calculate the precision of our model. When you are done, run the following cell to calculate the precision of our model.

```
[71]: from src.metrics import precision

      # Run this cell to calculate the precision
      precision_value = precision(con_matrix)

      # Display the precision
      print(f"Precision of the model on the test set: {precision_value:.3f}")
```

```
Precision of the model on the test set: 0.540
```

Describe what you see in the confusion matrix and other metrics above. What do they tell us?

On the test set, the model achieves precision 0.54, specificity 0.948, and sensitivity 0.254, with accuracy 0.81 and prevalence 19%. The confusion matrix (TN=528, FP=29, FN=100, TP=34) shows the model is highly conservative: it excels at ruling out the condition (very few false positives), but it misses many true cases (low recall). This pattern is consistent with class imbalance and a default decision threshold near 0.5—because most patients are negative, the model can look "good" on accuracy and specificity even while failing to identify a large share of positives. The negative predictive value is strong (0.84), meaning predicted negatives are usually correct, but the low sensitivity indicates substantial under-detection of true positives.

3.2.3 `2.2.3 (10 Points)` ROC and PR Curves

As we learned in the previous assignment, logistic regression models (like the model we used above) calculate the predicted probability of an example belonging to the positive class (1). The metrics above provide valuable insights about the performance of the classifier, but they are limited by their dependence on what is called the model's `decision threshold`. The `decision threshold` is the probability threshold above which we predict an example as positive (1), and below which we classify an example as negative (0). In most machine learning frameworks, the default decision threshold is 0.5 (which was the case for the predictions we made above). Although these metrics are useful, their dependence on a single decision threshold of the model can be limiting.

To achieve a more comprehensive assessment of our model's performance that spans various decision thresholds, we can construct the **receiver operating characteristic (ROC) curve** and the **precision-recall (PR) curve**. These tools enable us to visualize and understand the trade-offs between different threshold settings, thus providing a more dimensional view of the model's classification abilities.

```
[72]: # Run this cell to calculate the predicted probability of the positive class␣
       ↪for each sample in the test set
      # No need to change anything in this cell
      test_probabilities = log_reg_model.predict_proba(test_features)[:,1]
```

Receiver Operating Characteristic (ROC) Curve A ROC curve is a graphical representation that depicts the diagnostic capabilities of a binary classification system across a spectrum of decision thresholds. It is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold levels. The curve provides a comprehensive picture of the trade-off between sensitivity (or TPR) and specificity (1 - FPR) as the decision threshold is adjusted.

<img src="https://assets-global.website-files.com/6266b596eef18c1931f938f9/647603c5a947b640e4b
<figcaption> For a particular threshold value, the ROC curve shows the corresponding TPR and F

A metric that is commonly utilized to summarize the overall performance of the classifier is the `AUROC` or Area under the Receiver Operating Characteristic Curve (sometimes called ROC AUC). A higher AUROC (closer to 1) indicates a better performing model, because it means that for a given FPR, the corresponding TPR is higher on average.

<img src="https://assets-global.website-files.com/6266b596eef18c1931f938f9/64760408cdd765c3f8e
<figcaption> The area under the ROC curve </figcaption>

Let's think about this a little more: A perfect Receiver Operating Characteristic (ROC) curve would show a sharp rise from the origin (0,0) to the top-left corner of the plot (0,1) and then run directly along the top edge of the graph to the top-right corner (1,1). This shape indicates that the classifier is able to achieve a 100% true positive rate (TPR) without incurring any false positives. In this case, the area under the curve (AUROC) would be 1.0, which is the maximum value possible.

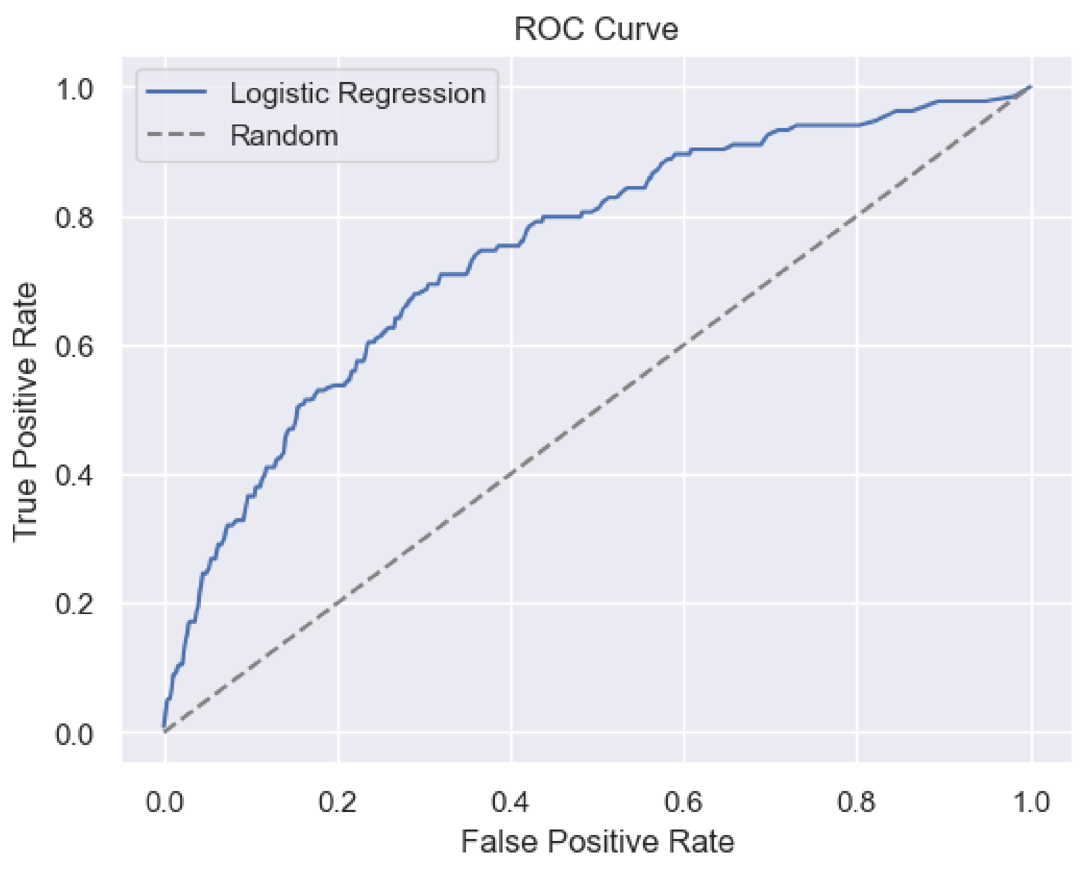
On the other hand, a ROC curve for a classifier that performs no better than random guessing would be a diagonal line that runs from the bottom-left corner (0,0) to the top-right corner (1,1) of the ROC space. This line is often referred to as the line of no-discrimination and it indicates that the classifier's true positive rate equals its false positive rate at all thresholds. In this case, the area under the curve (AUROC) would be 0.5.

<img src="https://assets-global.website-files.com/6266b596eef18c1931f938f9/647607123e84a06a426
<figcaption> The performance of three different models visualized with three curves. </figcapt

Lets see what our model's ROC curve looks like! Implement the functions `calc_roc_curve` and `display_roc_curve` in `curves.py` to calculate the ROC curve for our model. When you are done, run the following cells to calculate the ROC curve for our model.

[73]:
```python
from src.curve import calc_roc_curve

# Run this cell to calculate the ROC curve
fpr, tpr = calc_roc_curve(test_labels, test_probabilities)

# Sanity Check: FPR and TPR
if fpr.shape[0] != 1000:
    warnings.warn("FPR has incorrect number of values", SanityCheck)

if tpr.shape[0] != 1000:
    warnings.warn("TPR has incorrect number of values", SanityCheck)
```

[74]:
```python
from src.curve import display_roc_curve

# Run this cell to display the ROC curve
display_roc_curve([(fpr, tpr)],["Logistic Regression"],"lr_roc.png")
```



[75]:
```python
# Lets determine the AUROC of our model
# Run this cell to import the auc function from sklearn
# No modifications needed in this cell
from sklearn.metrics import auc

# Run this cell to calculate the AUROC
auroc = auc(fpr, tpr)

# Display the AUROC
print(f"Area under the ROC curve: {auroc:.3f}")
```

```
Area under the ROC curve: 0.746
```

Precision-Recall (PR) Curve Another comprehensive way to assess the performance of a classifier is a precision-recall (PR) curve, which is created by plotting recall on the x-axis and precision on the y-axis for different threshold values.

PR curves are more informative than ROC curves when dealing with highly imbalanced datasets, as the ROC curve may present an overly optimistic view of the performance.

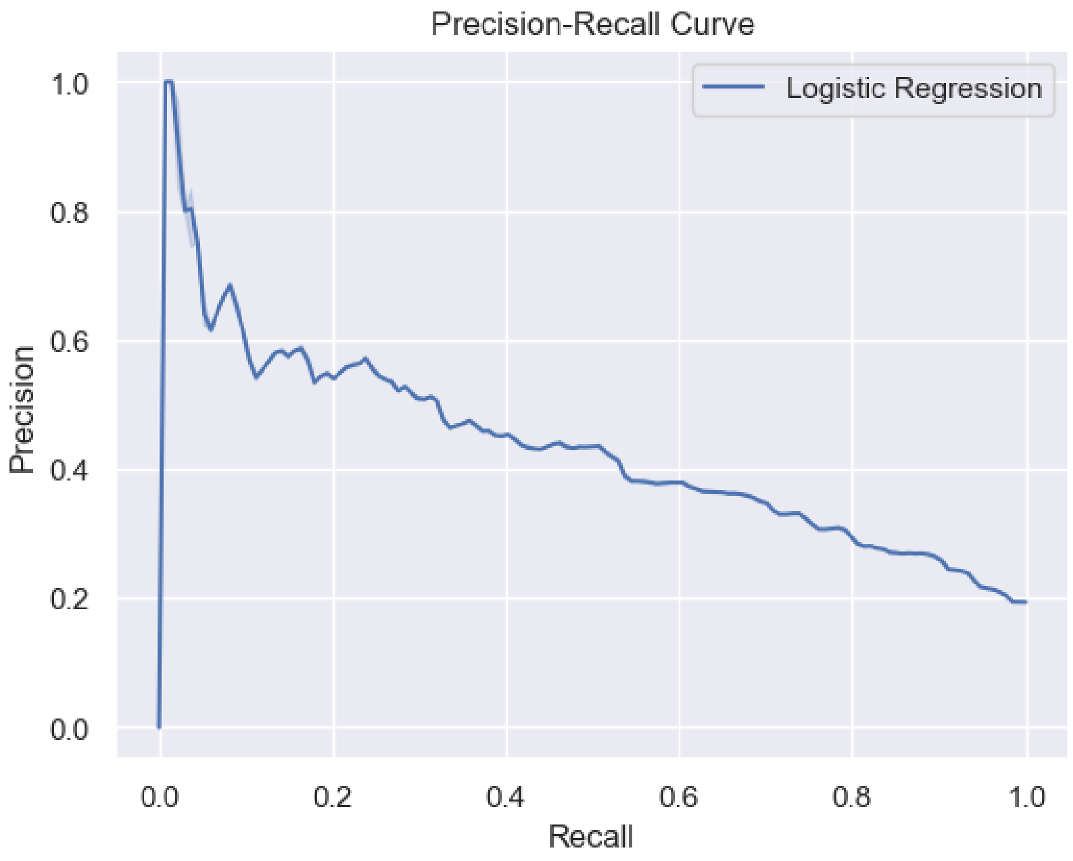
- In an imbalanced dataset, the number of negative instances (the majority 0 class) leads to an extremely large number of True Negatives, as the model can simply learn to predict 0 for almost every example. This makes the false positive rate ($\frac{FP}{FP+TN}$) appear to be very small over a wide range of thresholds, which can make the model appear to perform better on the ROC curve even if it is merely classifying most instances as the majority class (similar to our example above where the model always predicted 0).

- The precision-recall curve swaps out the FPR term with precision, also known as the positive predictive value ($\frac{TP}{TP+FP}$). Since the precision term does not take into account true negatives, it is not affected by the class imbalance problem described above and can give a better sense of the model's performance in terms of the cost of false positives.

The precision-recall curve is a bit more difficult to interpret than the ROC curve, but it is still a useful tool for evaluating the performance of a classifier. A metric that is commonly utilized to summarize the overall performance of the classifier is the `AUPRC` or Area under the Precision-Recall Curve. A higher AUPRC (closer to 1) indicates a better performing model, because it means that for a given recall, the corresponding precision is higher on average.

[76]:
```python
from src.curve import calc_precision_recall_curve

# Run this cell to calculate the precision-recall curve
prec, recall = calc_precision_recall_curve(test_labels, test_probabilities)

# Sanity Check: FPR and TPR
if prec.shape[0] != 1000:
    warnings.warn("FPR has incorrect number of values", SanityCheck)

if recall.shape[0] != 1000:
    warnings.warn("TPR has incorrect number of values", SanityCheck)
```

[77]:
```python
from src.curve import display_precision_recall_curve

# Run this cell to display the ROC curve
display_precision_recall_curve([(prec, recall)], ["Logistic Regression"],␣
 ↪"lr_prc.png")
```



[78]:
```python
# Lets determine the AUPRC of our model
# No modifications needed in this cell
auprc = auc(recall, prec)

# Display the AUPRC
print(f"Area under the precision-recall curve: {auprc:.3f}")
```

```
Area under the precision-recall curve: 0.431
```

3.2.4 2.2.4 (5 Points) Calibration Plot

In a calibration plot, the x-axis represents the predicted probabilities (often binned into intervals) that a model assigns to the positive class, while the y-axis represents the actual fraction of positives—the proportion of true outcomes that were correctly predicted within those bins.

The ideal outcome is that the predicted probabilities match the observed frequencies. If a model is perfectly calibrated, the points on the plot will fall on the diagonal line extending from the bottom left corner to the top right corner, known as the line of perfect calibration.

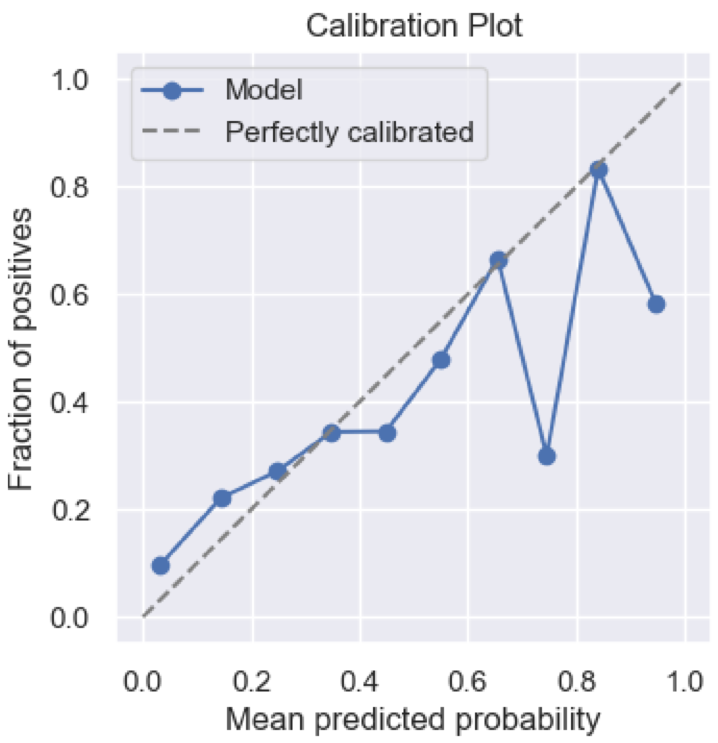
When the points lie above the diagonal, the model is said to be underconfident, meaning that it assigns lower probabilities to the positive outcomes than is justified by the actual outcome.

Conversely, when the points are below the diagonal, the model is overconfident, giving higher probabilities than warranted.

Calibration plots are essential for understanding whether you can trust the probabilistic predictions of a model in practical decision-making scenarios. If a model says an event has a 70% chance of happening, you want that event to actually occur approximately 70% of the time when the model makes such a prediction. Calibration plots help verify that the predicted probabilities of a model are aligned with the real-world frequencies.

Implement the function `display_calibration_plot` in `curve.py`. When you are done, run the following cell.

[79]:
```
# Run the following cell to display the calibration plot
from src.curve import display_calibration_plot

display_calibration_plot(test_labels, test_probabilities, "en_calibration_curve.
↪png")
```



Dicuss what you observe in the calibration plot above.

In the low-to-mid range (roughly 0–0.5), points track the diagonal fairly well and often sit slightly above it, indicating mild underconfidence: the model's probabilities are a bit lower than the empirical event rates. As probabilities increase beyond 0.6, the curve becomes more erratic. Some mid-high bins remain above the diagonal (still underconfident), but others—especially near the top of the score range—fall below the diagonal, signaling overconfidence: for these high-confidence predictions, the observed outcome rate is notably lower than predicted. This zig-zag behavior at high scores likely reflects limited sample sizes and class imbalance within those bins, which introduces variance in the estimated fraction of positives. Practically, predicted probabilities are more trustworthy in the low-to-mid ranges and should be interpreted cautiously at the high end.

3.3 `2.3 (30 Points)` Hyperparameter tuning and Cross-Validation

It is impossible to know in advance what the best value of λ is for a given model on a given dataset. One way to find a good value is to try training the model with many different values of λ and see which one works best - this is known as *hyperparameter tuning*. Let's try it.

3.3.1 `2.3.1 (10 Points)` Cross-Validation

Cross validation is a statistical method used for evaluating and comparing machine learning models on a limited data sample. It is often employed to estimate the performance of a model on unseen data, which can be helpful when we need to perform hyperparameter tuning.

The most common method of cross validation is the **k-fold cross validation**. Here's how it works:

1. First, we partion our training data into k equal-sized randomly partitioned subsamples.
2. Of the k subsamples, k-1 subsamples are used to fit the model, and the remaining subsample is used to evaluate the model's performance.
3. The cross-validation process is then repeated k times, with each of the k subsamples used exactly once for evaluation.
4. The k results from the folds can then be averaged (or otherwise combined) to produce a single estimation of the overall model's performance.

```
<img src="https://scikit-learn.org/stable/_images/grid_search_cross_validation.png" style="bac
<figcaption> A visualization of cross validation </figcaption>
```

Let's perform a 5-fold cross validation on our model to see how different values of λ affect the performance of our model.

[80]:
```python
# First, let's make a range of lambda values to try.
# Run this cell: No need to change anything
lambdas = np.exp(np.arange(-11, -1.9, 1))

print(f"Lambda values to try: {lambdas}")
print(f"Number of lambda values to try: {len(lambdas)}")
```

```
Lambda values to try: [1.67017008e-05 4.53999298e-05 1.23409804e-04
3.35462628e-04
 9.11881966e-04 2.47875218e-03 6.73794700e-03 1.83156389e-02
 4.97870684e-02 1.35335283e-01]
Number of lambda values to try: 10
```

Implement the function `logistic_regression_cv` in `cross_val.py`. When you are done, run the following cell.

```
[81]: from src.cross_val import logistic_regression_cv

      # Run this cell to perform cross-validation
      # NOTE: This may take a few minutes to run (It took about 1.5 minutes on the␣
       ↪TA's computer)
      # NOTE: You may see some warnings about convergence. This is ok!

      # In this assignment, we will ignore convergence warnings
      log_reg_cv = logistic_regression_cv(train_features, train_labels, 4, lambdas,␣
       ↪max_iter=30000)
```

```
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
```

```
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/opt/miniconda3/envs/biomedin215/lib/python3.11/site-
packages/sklearn/linear_model/_logistic.py:460: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of f AND g EVALUATIONS EXCEEDS LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

Let's take a look at the average AUROC for each value of λ we tried.

[82]:
```
# Display the mean AUROC values
print("Mean AUROC values:")
print(log_reg_cv.cv_results_["mean_test_score"])
```

```
Mean AUROC values:
[0.73859305 0.75315398 0.74204927 0.74148746 0.74994666 0.74731894
 0.74494723 0.75578881 0.74389472 0.75140098]
```

3.3.2 2.3.2 (5 Points) Model Performance

Now that we have the cross-validation results, let's use the lambda of the best model to train a new model on the full training set and evaluate it on the test set. Determine the AUROC and AUPRC values and report them in the answer space below.

[83]:
```
# Collect the best lambda value from the cross-validation results
import numpy as np
from src.models import logistic_regression
from src.curve import calc_roc_curve, calc_precision_recall_curve

best_C = log_reg_cv.best_params_["C"]
best_lambda_val = 1.0 / best_C

# Fit a new model with the best lambda value
best_lambda_model = logistic_regression(
    train_features, train_labels, lambda_=best_lambda_val, random_state=42,␣
 ↪max_iter=10000
)

# Calculate probabilities using the new model
best_lambda_probabilities = best_lambda_model.predict_proba(test_features)[:, 1]

# Calculate the ROC curve
best_lr_fpr, best_lr_tpr = calc_roc_curve(test_labels,␣
 ↪best_lambda_probabilities)

# AUROC: flip arrays so x is increasing to avoid negative area
best_lr_auroc = float(np.trapz(y=np.flip(best_lr_tpr), x=np.flip(best_lr_fpr)))

# Calculate the precision-recall curve
best_lr_prec, best_lr_recall = calc_precision_recall_curve(test_labels,␣
 ↪best_lambda_probabilities)

# AUPRC: flip arrays so x (recall) is increasing
best_lr_auprc = float(np.trapz(y=np.flip(best_lr_prec), x=np.
 ↪flip(best_lr_recall)))
```

[84]:
```
# Display the results from the above cell
# Run this cell: No need to change anything
if best_lambda_val is not None:
    print(f"Best value of lambda: {best_lambda_val:.4f}")

if best_lr_auroc is not None:
    print(f"Best AUROC: {best_lr_auroc:.3f}")

if best_lr_auprc is not None:
    print(f"Best AUPRC: {best_lr_auprc:.3f}")
```

```
Best value of lambda: 0.0183
Best AUROC: 0.744
Best AUPRC: 0.425
```

3.3.3 2.3.3 (5 Points) Test Error Estimation After Feature Selection

Zuko from Fire Nation University has also been attempting to implement a logistic regression model to predict mortality in this cohort. Before training (and without looking at the test data), Zuko performed univariate analyses to find all features that had a statistically significant association with mortality in the training set, and then he only used those features to fit his models.

Would you expect that the average AUROC values achieved by the model during the cross-validation would be on average larger, smaller, or the same as the AUROC Zuko's model would achieve on the test set? Why?

Larger. Due to the fact that he is only using features that are highly associated with the outcome (mortality) in the test set, will essentially mean his model will be optomize for the testing data nad out perform. Zuko screens features by testing many univariate associations on the training data and then fits a model on only the "significant" ones. That selection step uses outcome information and overfits, so the apparent signal shrinks on truly unseen data, lowering his test AUROC.

3.3.4 2.3.4 (5 Points) Comparing Feature Selection Methods

Is Zuko's approach to feature selection the same as what we did earlier by removing the near-zero-variance features in the sense that we should we expect it to have a similar effect on the test vs. cross-validation AUC? Why or why not?

No, it is not. Removing these near-zero-variance features is an unsurpervised, outcome agnostic, filtering done before modeling. Zuco uses the outcome/targets in his filtering, augmenting his model with information it should be siloed from. Removing zero-variance uses only feature distributions (not the outcome), so it doesn't leak label information and mainly reduces noise/variance; its CV vs. test AUC should be similar (sometimes slightly better due to reduced overfit). The leakage of outcome information will make Zuco's model overly optimisitc. We do not see this problem with the removal of zero-variance features.

3.3.5 2.3.5 (5 Points) Inspecting Coefficients

Let's check out the 10 features in our model with the largest coefficients. Implement the functions `get_top_features` and `get_log_reg_coefficients` in `models.py`. When you are done, run the following cell.

[85]:
```python
from src.models import get_top_features

# Run this cell to get the top features
top_features = get_top_features(best_lambda_model, filtered_features,␣
 ↪feature_descriptions, model_type="logistic_regression")
```

[86]:
```python
# Display the top features
# Run this cell: No need to change anything
display(top_features)
```

```
              feature    feature_type  \
0   chartindicator_50  chartindicator
1      chartvalue_829      chartvalue
2   chartindicator_54  chartindicator
```

```
3  chartindicator_623  chartindicator
4  chartindicator_786  chartindicator
5  chartindicator_604  chartindicator
6            oxy_drop      engineered
7  chartindicator_704  chartindicator
8  chartindicator_785  chartindicator
9            C0032227        note CUI

                                       description      coef  abs_coef
0                             Apnea Time Interval  0.228593  0.228593
1                             Potassium (3.5-5.3)  0.220254  0.220254
2                               Assistance Device  0.218143  0.218143
3                            Restraints Evaluated  0.204797  0.204797
4                              Calcium (8.4-10.2) -0.203499  0.203499
5                                Radiologic Study -0.194911  0.194911
6  sustained drop in oxygenation after stable ven…  0.194152  0.194152
7                                            Turn -0.193861  0.193861
8                                          CPK/MB -0.188412  0.188412
9                                pleural effusion  0.181488  0.181488
```

3.4 2.4 (30 Points) Gradient Boosted Trees

Let's see if we can get better performance by utilizing a different type of model: gradient boosted trees.

A **decision tree** is a tree structured model where internal nodes represent conditional tests that operate on features of the data, and each leaf node represents a potential final decision or prediction (in this case, the predicted class label). Let's take a look at a simple three layer example of a decision tree:

<img src="https://www.baeldung.com/wp-content/uploads/sites/4/2022/02/1_decision_tree1.jpg" wi
<figcaption> A simple three layer decision tree </figcaption>

In the image above, let's say the task was to predict if today is a good day to go for a bike ride. First, the tree checks if the humidity feature is either high or low. If Humidity is low, the tree will immediately output the prediction "Yes". If Humidity is high, the tree will check the Weather feature to see if the conditions are "sunny", "overcast", or "rainy". This process repeats until the tree reaches a leaf node, at which point it outputs the prediction.

By themselves, decision trees are considered weak predictive models that are prone to overfitting. This is where gradient boosting comes in.

Gradient-boosting iteratively trains a series of decision trees, where each subsequent tree is fit on the residual errors of the previous trees. In the context of binary classification, the residual error is the difference between the predicted probability of the positive class and the actual label. At each stage, the model fits a new decision tree to the residual errors of the previous tree, and then adds the new tree to the ensemble of trees that came before it. The final prediction is the sum of the predictions of all the trees in the ensemble.

Tree based methods are powerful for clinical prediction tasks because of their simplicity and interpretability. They mimic human decision-making processes more closely than other algorithms, making them easy to understand and visualize. This interpretability is often a significant advantage in medicine, as understanding the model's decision-making process is often crucial to ensure that the model is safe and effective.

3.4.1 `2.4.1 (5 Points)` Parameters for Gradient Boosted Trees

Let's see if a gradient boosted tree can outperform our logistic regression model. To begin, let's first explore the parameters of the gradient boosted tree model we will be using.

Read the documentation of the sklearn.ensemble.GradientBoostingClassifier.

Let's think about how various parameters may impact the "bias" and "variance" tradeoff we learned about in A5. Remember that (in the context of bias vs. variance):

- **Bias** refers to the difference between the average prediction of our model and the correct value we are trying to predict. A model is said to have high bias if the model oversimplifies the "true" relationship between the predictors and outcome, generally referred to as "being underfit". Models with high bias often have systematic errors in predictions (errors largely occuring in the same direction with similar magnitude) as the estimates are "biased" away from the true values.
- **Variance** refers to the variability of our model's predictions. A model is said to have high variance if small changes to the training data result in large changes in the predictions. Models with high variance are often **overfit** to the training data, and do not generalize well to new data.

For each of the following parameters, please describe what it does and whether it increases or decreases the bias and variance.

---

n_estimators Number of trees added sequentially. More trees let the ensemble fit residuals more closely. Effect: decreases bias, increases variance and training time. Too many trees can overfit unless learning_rate is small or early stopping is used.

---

max_depth Maximum depth of each individual tree. Deeper trees capture higher-order interactions. Effect: decreases bias as high order-interactions are more complicated and specific. Increases variance as larger branching factors increases outcome possibilities; shallow trees increase bias but stabilize/generalize better.

---

learning_rate Learning rate is the shrinkage factor which you multiply each tree's contribution by. Tradeoff with n_estimators. A lower learning_rate will increase bias per step. Because each tree's effect is shrunk, you are likely to underfit the model unless there are a very large number of trees. Higher learning rate decreases bias but risks higher variance. Meanwhile, a lower learning rate will decrease variance. Each step is extremely regulated, preventing large swings and overfitting. At the same time, higher learning rate increase variance due to a large propsensity for swings and overfitting.

min_samples_split and min_samples_leaf

> Note: These two are pretty similar and can be answered together (they have the same effect on bias and variance).

Minimum samples to split a node / to keep in a leaf. Larger values restrict growth, prune small splits, and smooth predictions. This will increase bias and reduce variance by averaging over large splits and limitting noisy leaves.

3.4.2 2.4.2a (2 Points) Setting up a Gradient Boosted Tree Model

Let's set up a cross validation experiment to find the best parameters for our gradient boosted tree model. Implement the function `gradient_boosted_tree` in `models.py`. When you are done, run the following cell.

[87]:
```python
from src.models import gradient_boosted_tree

# Run this cell to fit the model
gbt_model = gradient_boosted_tree(train_features, train_labels,␣
 ↪n_estimators=100)
```

[88]:
```python
# Let's see how this model performed
# Run this cell to see the accuracy, precision, and recall
# NOTE: No need to change anything in this cell

test_predictions = gbt_model.predict(test_features)

# Calculate the accuracy
gbt_accuracy = accuracy(test_labels, test_predictions)

# Calculate the confusion matrix
gbt_con_matrix = confusion_matrix(test_labels, test_predictions)

# Calculate the sensitivity
gbt_sensitivity = sensitivity(gbt_con_matrix)

# Calculate the specificity
gbt_specificity = specificity(gbt_con_matrix)

# Calculate the precision
gbt_precision = precision(gbt_con_matrix)

# Display the results
print(f"Accuracy of the model on the test set: {gbt_accuracy:.3f}")
print(f"Sensitivity of the model on the test set: {gbt_sensitivity:.3f}")
print(f"Specificity of the model on the test set: {gbt_specificity:.3f}")
print(f"Precision of the model on the test set: {gbt_precision:.3f}")
```

```
Accuracy of the model on the test set: 0.821
Sensitivity of the model on the test set: 0.201
Specificity of the model on the test set: 0.969
Precision of the model on the test set: 0.614
```

3.4.3 2.4.2b (3 Points) Cross-Validating Gradient Boosted Trees

Let's set up a cross validation experiment to find the best parameters for our gradient boosted tree model. Implement the function `gradient_boosting_cv` in `cross_val.py`. When you are done, run the following cell.

[89]:
```
# Let's try a range of n_estimators values
# Run this cell: No need to change anything
n_estimators_list = np.arange(0, 201, 5)
n_estimators_list[0] = 2

print(f"We are trying {len(n_estimators_list)} values of n_estimators:␣
 ↪{n_estimators_list}")
```

```
We are trying 41 values of n_estimators: [  2   5  10  15  20  25  30  35  40
45  50  55  60  65  70  75  80  85
  90  95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175
 180 185 190 195 200]
```

[90]:
```
from src.cross_val import gradient_boosting_cv

# Run this cell to perform cross-validation
# NOTE: This cell may take a few minutes to run (It took about 1.5 minutes on␣
 ↪the TA's computer)
gb_cv = gradient_boosting_cv(train_features, train_labels, 4, n_estimators_list)
```

3.4.4 2.4.3 (5 Points) Model Performance

Let's see how well our model performs on the test set with the best parameters we found during cross-validation. Determine the AUROC and AUPRC values.

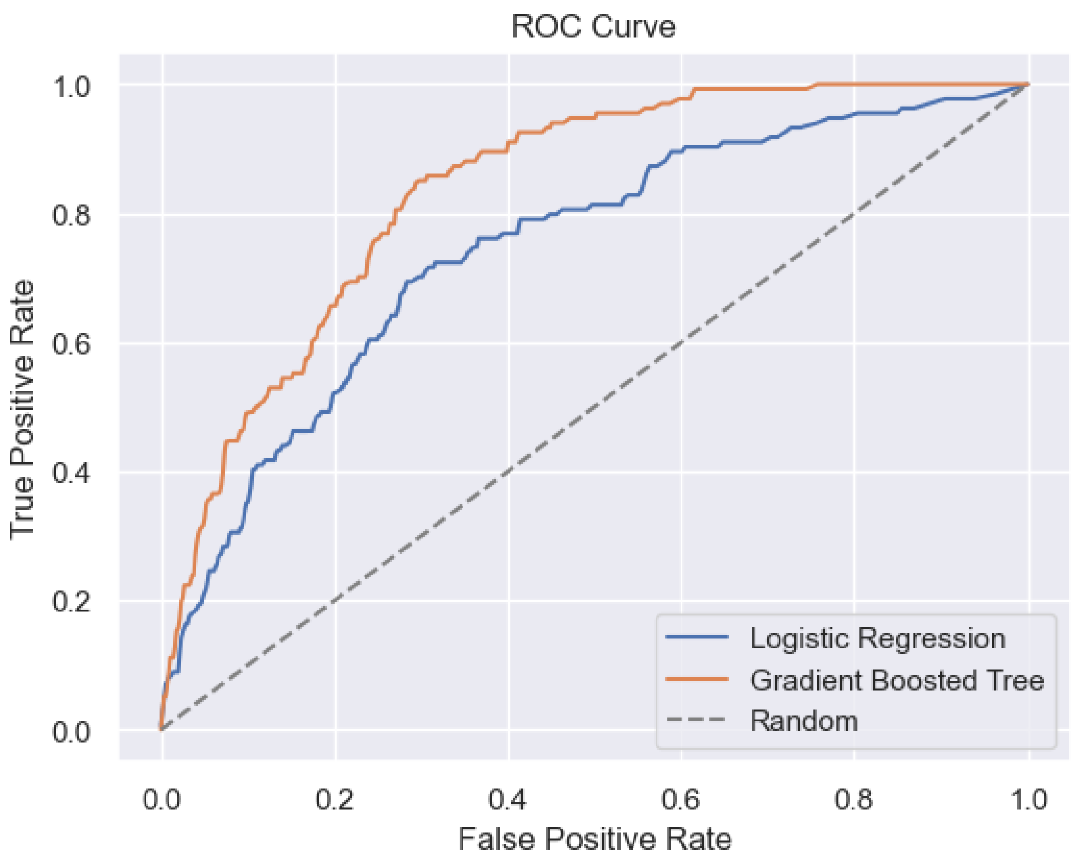
[91]:
```
# Use this cell to train a model with the best n_estimators value from the␣
 ↪cross-validation

import numpy as np
from src.models import gradient_boosted_tree
from src.curve import calc_roc_curve, calc_precision_recall_curve

# Collect the best n_estimators value from the cross-validation results
best_n_estimators_val = int(gb_cv.best_params_["n_estimators"])

# Fit a new model with the best n_estimators value
best_tree_model = gradient_boosted_tree(
    train_features, train_labels, n_estimators=best_n_estimators_val,␣
 ↪random_state=42
)

# Calculate the probabilities using the new model
test_probabilities = best_tree_model.predict_proba(test_features)[:, 1]

# Calculate the ROC curve using your calc_roc_curve function
best_tree_fpr, best_tree_tpr = calc_roc_curve(test_labels, test_probabilities)

# Calculate the new AUROC (flip so x increases)
best_tree_auroc = float(np.trapz(y=np.flip(best_tree_tpr), x=np.
 ↪flip(best_tree_fpr)))

# Calculate the new precision-recall curve using your␣
 ↪calc_precision_recall_curve function
best_tree_prec, best_tree_recall = calc_precision_recall_curve(test_labels,␣
 ↪test_probabilities)

# Calculate the new AUPRC (flip so x increases)
best_tree_auprc = float(np.trapz(y=np.flip(best_tree_prec), x=np.
 ↪flip(best_tree_recall)))
```

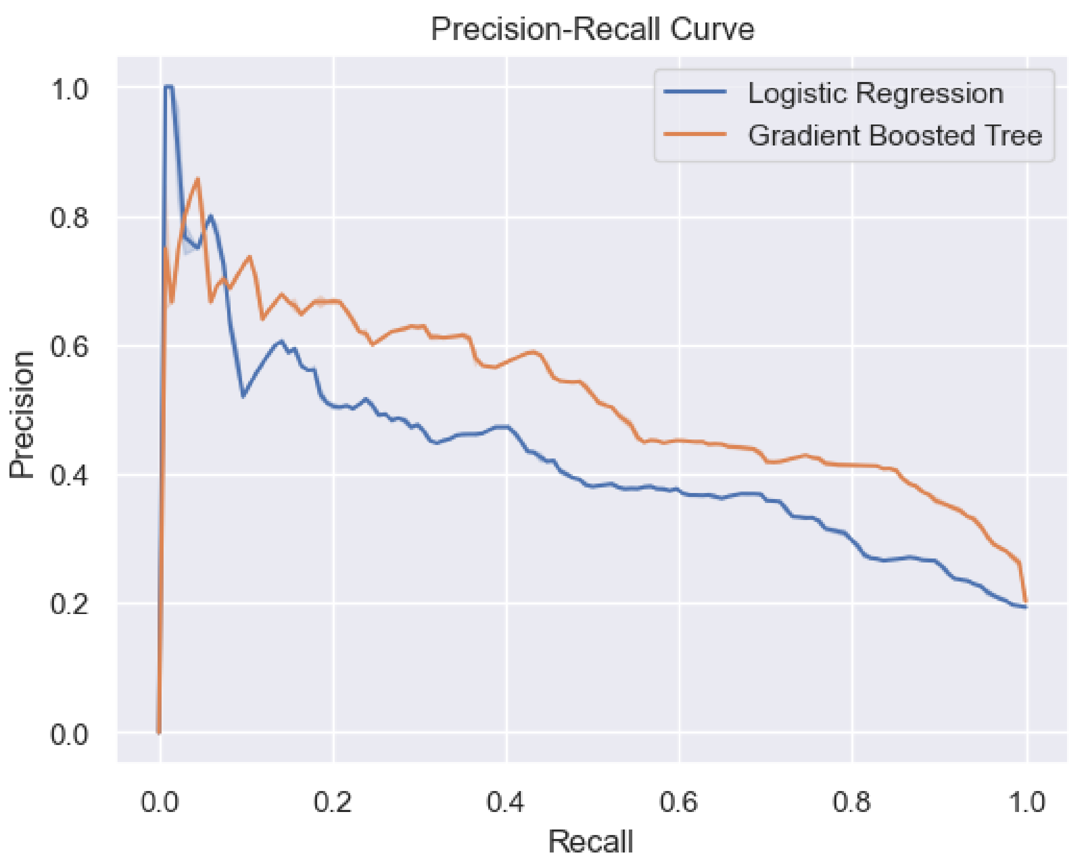
[92]:
```
# Display the results from the above cell
# Run this cell: No need to change anything
if best_n_estimators_val is not None:
    print(f"Best value of best_n_estimators_val: {best_n_estimators_val:.4f}")

if best_tree_auroc is not None:
    print(f"Best AUROC: {best_tree_auroc:.3f}")

if best_tree_auprc is not None:
    print(f"Best AUPRC: {best_tree_auprc:.3f}")
```

```
Best value of best_n_estimators_val: 60.0000
Best AUROC: 0.837
Best AUPRC: 0.521
```

Finally, let's take a look at the difference between the curves!

[93]:
```
# Run this cell! No need to change anything
if best_lr_fpr is not None and best_tree_prec is not None:
    display_roc_curve([(best_lr_fpr, best_lr_tpr), (best_tree_fpr,␣
 ↪best_tree_tpr)],
                     ["Logistic Regression", "Gradient Boosted Tree"],
                     "lr_gbt_roc.png")
    display_precision_recall_curve([(best_lr_prec, best_lr_recall),␣
 ↪(best_tree_prec, best_tree_recall)],
                                     ["Logistic Regression", "Gradient Boosted␣
 ↪Tree"],
                                     "lr_gbt_prc.png")
```





3.4.5 2.4.4 (5 Points) Variable Importance

Let's determine the top 10 most important features in our model, using the function `get_tree_feature_importance` that you previously implemented in `models.py`, by running the following cell.

```
[94]: # Run this cell to get the top features
top_10_tree_features = get_top_features(best_tree_model, filtered_features,␣
  ↪feature_descriptions, model_type="gradient_boosted_tree")
```

```
[95]: # Display the top features
# Run this cell: No need to change anything
display(top_10_tree_features)
```

```
              feature    feature_type           description  importance  \
0      chartvalue_198      chartvalue             GCS Total    0.088443
1         age_in_days     demographic           age in days    0.086778
2      chartvalue_778      chartvalue        Arterial PaCO2    0.051097
3  chartindicator_655  chartindicator  Spontaneous Movement    0.043485
4      chartvalue_454      chartvalue        Motor Response    0.042967
5      chartvalue_781      chartvalue            BUN (6-20)    0.034200
```

```
6  chartindicator_412  chartindicator     Incision/Wound #2    0.032179
7  chartindicator_548  chartindicator       Position Change    0.032156
8       chartvalue_87      chartvalue          Braden Score    0.031823
9      chartvalue_777      chartvalue    Arterial CO2(Calc)    0.023534

   abs_importance
0        0.088443
1        0.086778
2        0.051097
3        0.043485
4        0.042967
5        0.034200
6        0.032179
7        0.032156
8        0.031823
9        0.023534
```

Let's learn about how feature importance is calculated in `sklearn`. Read the following documentation sections: 1. Interpretation with Feature Importance

2. 1.11.2.5. Feature importance evaluation

Would a variable that is only split on in the first tree be more important than a variable that is only split on in the 200th tree?

In GradientBoostingClassifier, feature_importances_ is the normalized total impurity reduction a feature achieves across all trees. It is not inherently weighted by the stage index. So, a split in the first tree is not automatically more important than an equal-quality split in the 200th tree. In practice, earlier trees often reduce impurity more because they address larger residuals, so they may contribute more—but that's due to larger reductions, not "being early."

Which types of feature seem to be the most useful? Why do you think that is the case?

the most useful featuers are physiological chart values and a few high-sgnal nuerologic indicators along with age. GCS Total and Motor Response capture neurologic status; lower scores strongly track mortality risk. Arterial PaCO2 and calculated CO2 reflect ventilatory failure and acid–base status—abnormal values indicate inadequate ventilation or severe disease. Bun, incision, and Braden score are also quite relevant. These features are powerful because they are high-coverage, measured near the prediction time, and have relatively monotonic relationships with mortality, likely giving the tree clear, low-variance splits. Sparse or noisier features will likely rank lower because they add variance and less consistent signal.

3.4.6 `2.4.5 (5 Points)` Partial Dependence Plots

One of the nice things about tree ensembles is that they can automatically find and exploit interactions between features. One of the ways we can visualize these interactions is by utilizing partial dependence plots.

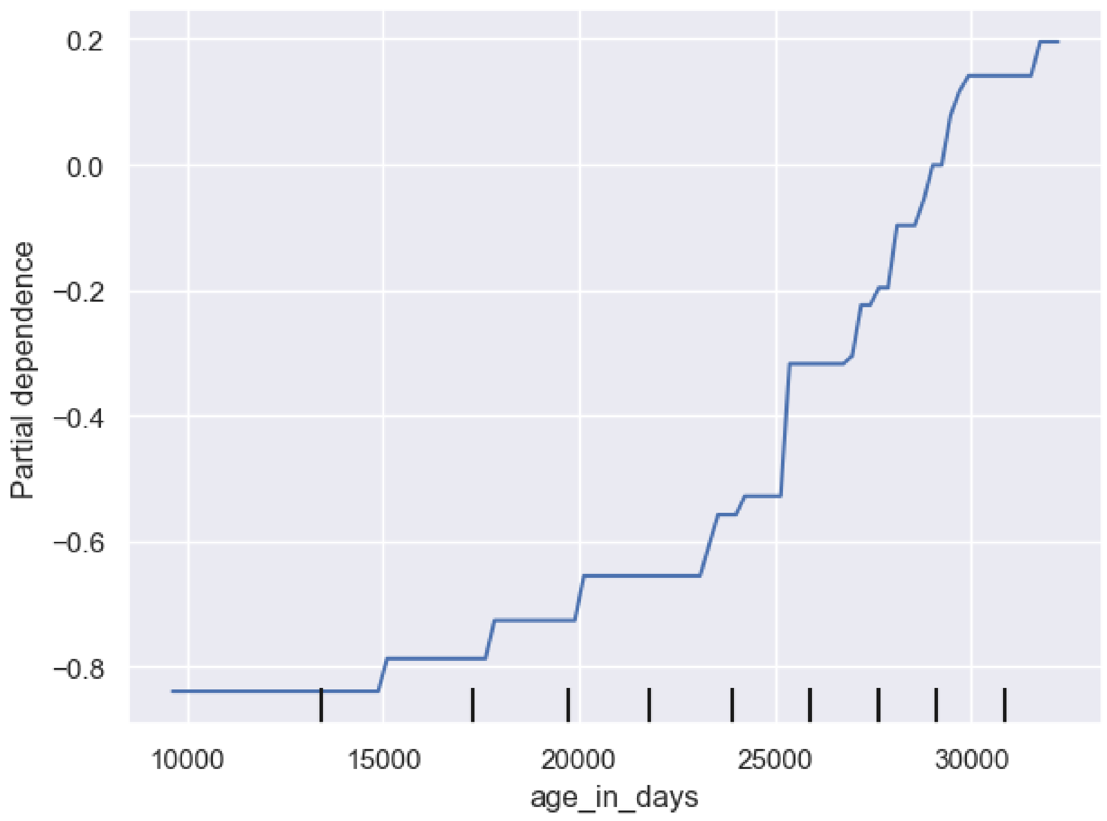
A partial dependence plot (PDP) visualizes the relationship between a subset of features and the predicted outcome of a machine learning model, averaged over the distribution of the other features in the dataset. In other words, it shows the effect of a feature (or features) on the prediction while holding other features constant.

A negative partial dependence value for a particular feature value indicates that feature value has a negative effect on the prediction, while a positive partial dependence value indicates that feature value has a positive effect on the prediction. A partial dependence value of 0 indicates that the feature value has no effect on the prediction (holding other features constant).

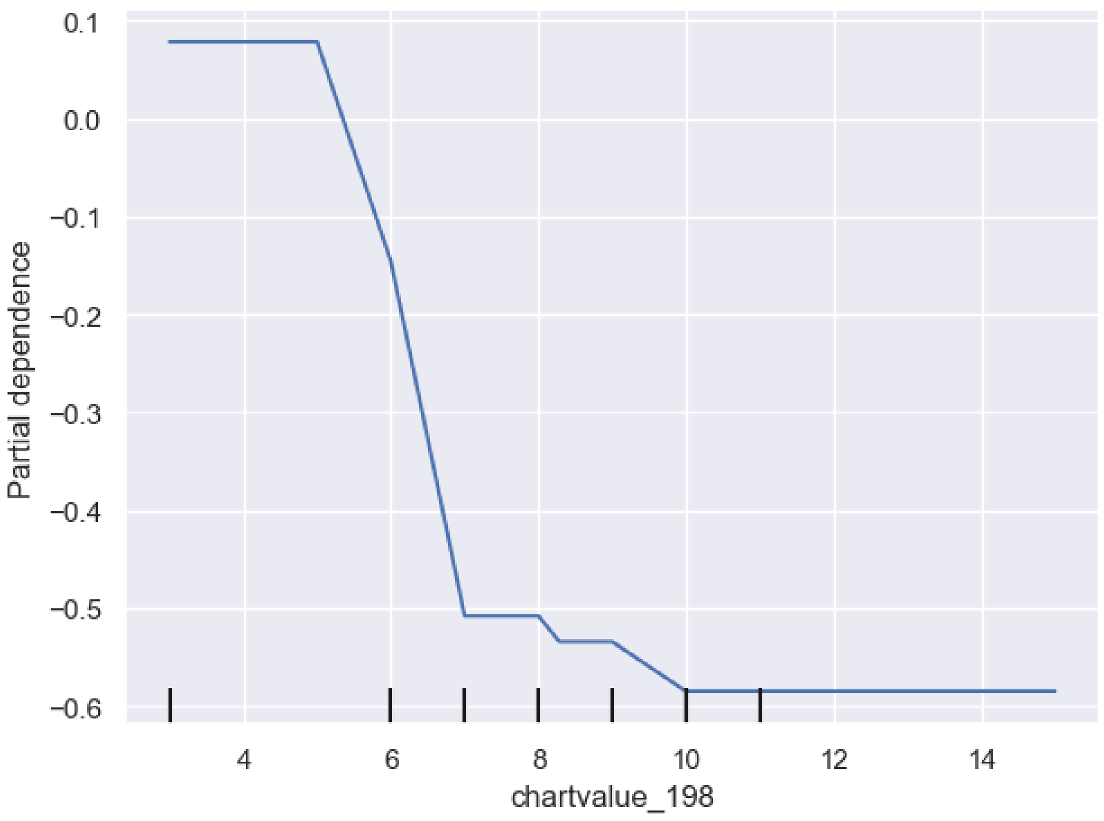
Implement the function `display_partial_dependence` in `inspection.py`. When you are done, run the following cell.

[96]:
```
from src.inspection import display_partial_dependence

# Run this cell to plot the partial dependence of age (by itself)
display_partial_dependence(best_tree_model, filtered_features, ["age_in_days"],␣
 ↪"gbt_age_pdp.png")

# Run this cell to plot the partial dependence of `chartvalue_198` aka GCS␣
 ↪total (by itself)
display_partial_dependence(best_tree_model, filtered_features,␣
 ↪["chartvalue_198"], "gbt_gcs_pdp.png")

# Run this cell to plot the partial dependence of age and GCS total
display_partial_dependence(best_tree_model, filtered_features, [("age_in_days",␣
 ↪"chartvalue_198")], "gbt_age_gcs_pdp.png")
```

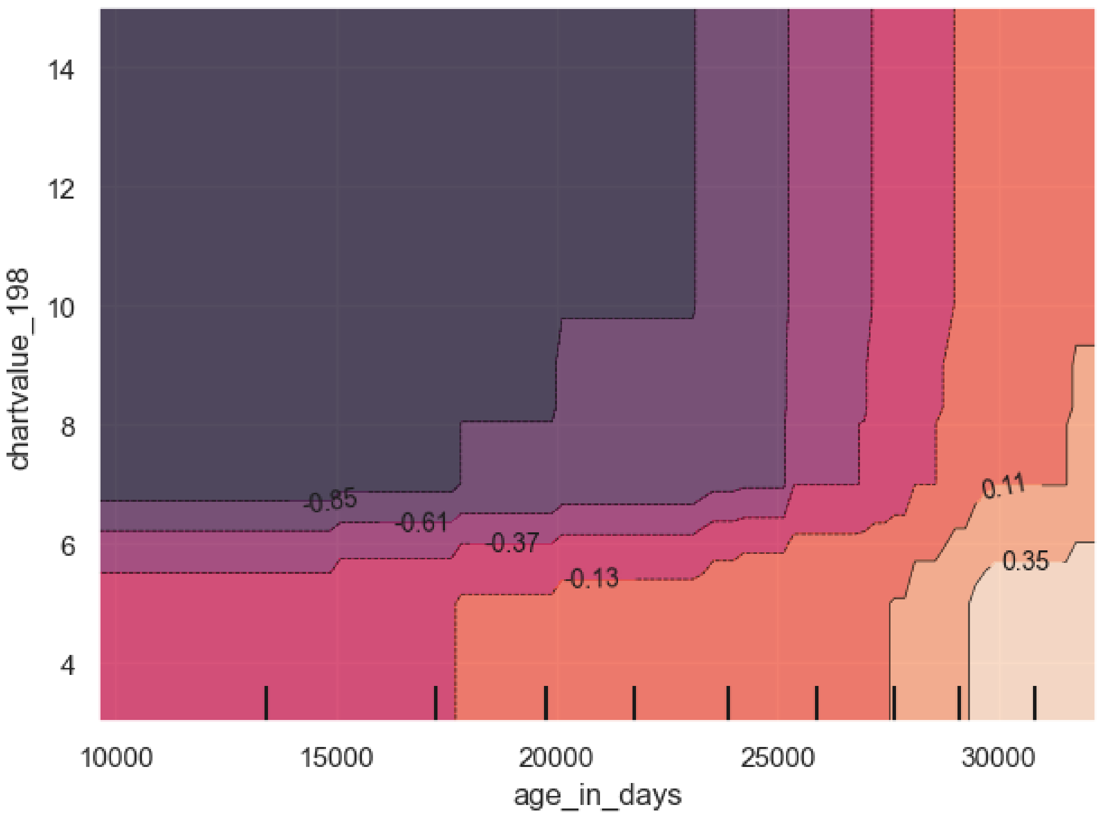
```
<Figure size 640x480 with 0 Axes>
```


0.2
0.0
−0.2
−0.4
−0.6
−0.8
Partial dependence
10000
15000
20000
25000
30000
age_in_days

```
<Figure size 640x480 with 0 Axes>
```


Partial dependence
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
4
6
8
10
12
14
chartvalue_198

```
<Figure size 640x480 with 0 Axes>
```



Is the effect of age on the model's predictions linear?

No, the effect on age appears to be exponential. This makes sense as mortality risk is far high at older ages than at lower ages, reflecting exponential increasing possibility of death.

What combination of these two features is most associated with worse outcomes? Do you think the result makes sense? Why or why not?

> This article on the Glasgow Coma Scale may be helpful!

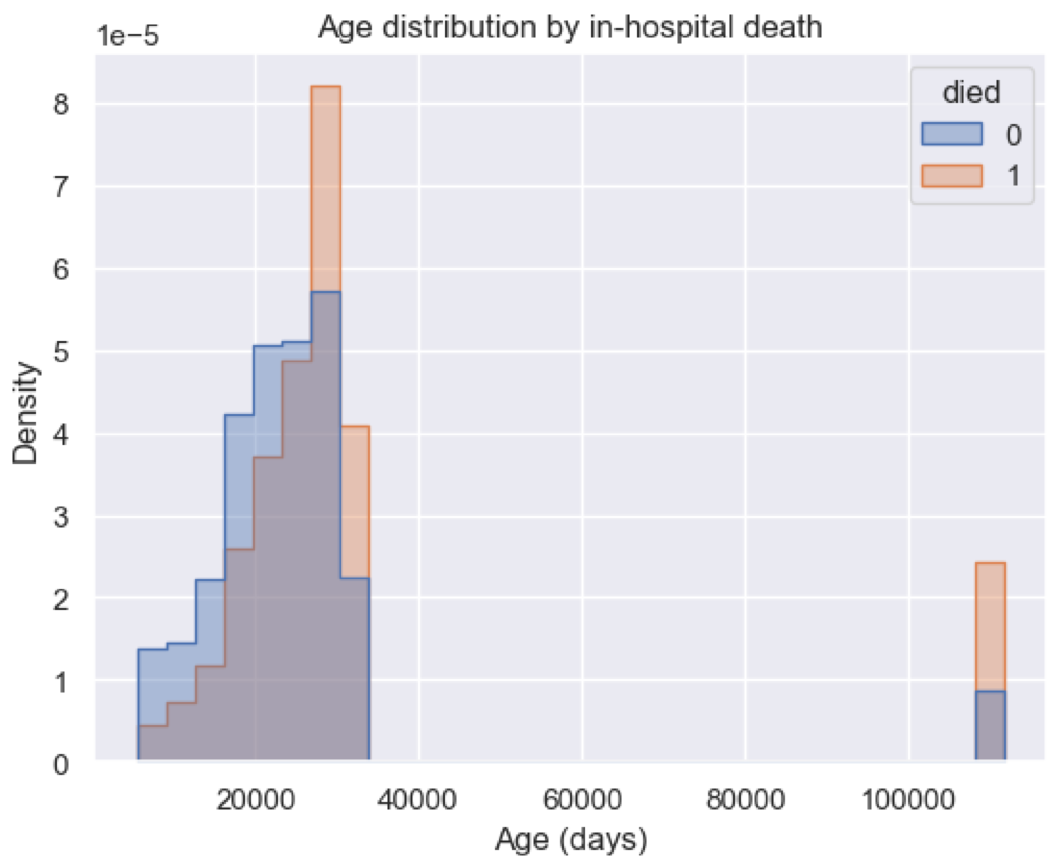
The 2D partial dependence shows the worst outcomes for patients who are both older and have low GCS Total. Risk rises sharply once GCS falls below about 6–7, and this effect is amplified with age: the darkest/red region is at high age_in_days combined with GCS 6. Even at moderate GCS (7–9), predicted risk increases with age, whereas younger patients with high GCS (10–12) have the lowest risk. This pattern makes clinical sense: low GCS reflects impaired consciousness/neurologic injury (GCS '<'8 classically signals severe injury and airway risk), and advanced age is a strong baseline risk factor due to frailty, comorbidity, and limited physiologic reserve.

3.4.7 `2.4.6 (5 Points)` Investigating the Effect of Age

Something is strange about the range of the age variable in the partial dependence plot above. Let's investigate this further.

Implement the function `display_age_distribution` in `inspection.py`. When you are done, run the following cell.

```
[97]: from src.inspection import display_age_distribution

      # Run this cell to display the age distribution
      display_age_distribution(filtered_features, labels, "gbt_age_dist.png")
```



What did you discover about the age variable?

Age is in days and shows a second spike around ~110,000 days (~300 years). That's the MIMIC de-identification rule: patients older than ~89 years are shifted to an extreme value. So the distribution is bimodal and includes implausible "very old" ages that actually encode "age   89".

Do you think this adversely affects the test set peformance of the gradient boosted tree model?

This will somewhat affect preformance. Trees will split on the artificial threshold and treat "age 89" as a strong rule, which can help within this dataset but harms calibration at the high end and reduces external validity.

What do you think the effect is on the logistic regression model? (HINT: Was `age_in_years` included in the top 10 features of the logistic regression model?)

Logistic regression won't capture the step at the de-ID cutoff; regularization will likely shrink its coefficient, which is why age_in_years often doesn't appear among the top coefficients.

3.5 `2.5 (5 Points)` Proper Evaluation of Predictive Models

Given that we randomly split the data into training and test sets, do you think that the test set accuracy would be a good estimate, an overestimate, or an underestimate of the accuracy if we used this model to predict mortality for patients in the coming year?

Justify your answer in one or two sentences. Give a suggestion for an alternative data-splitting method that could be better and why.

Overestimate. A random split draws train/test from the same time period and care environment, so it ignores temporal and practice drift. We can't capture a large number of variables that will actually exist in the practice next year which reflect year over year change in practice. As such, we will preform worse on data from the upcoming year.

4 Congratulations!

You have completed the final coding assignment of `BIOMEDIN 215`!

We hope you have enjoyed utilizing the Python version of the assignments in the course!

We would love to hear your thoughts on how we can improve the assignments for future students. If you haven't already, please fill out the feedback form below.

Feedback form

`- Your BIOMEDIN 215 Teaching Team`

---

5 Submission Instructions

There are two files you must submit for this assignment:

1. A `PDF` of this notebook.

- **Please clear any large cell outputs from executed code cells before creating the PDF.**
  - Including short printouts is fine, but please try to clear any large outputs such as dataframe printouts. This makes it easier for us to grade your assignments!
- To export the notebook to PDF, you may need to first create an HTML version, and then convert it to PDF.

2. A `zip` file containing your code generated by the provided `create_submission_zip.py` script:

- Open the `create_submission_zip.py` file and enter your SUNet ID where indicated.
- Run the script via `python create_submission_zip.py` to generate a file titled `<your_SUNetID>_submission_A6.zip` in the root project directory.