Modeling the Wisconsin Breast Cancer Dataset

Elliot Gorokhovsky

1 Introduction

In this project, I applied several methods from the course to the University of Wisconsin Breast Cancer Dataset, which I obtained from the UCI Machine Learning Repository. The dataset consists of 699 observations from breast cancer biopsies with 9 numerical features, corresponding to features of tumor biopsies observed under the microscope, and 1 binary feature corresponding to tumor malignancy. My goal was to predict whether a tumor is malignant from the visual features.

1.1 Features

Here is a brief overview of the columns of the dataset:

In [123]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import models, utils
```

Out[123]:
```
<module 'utils' from '/home/elliot/ml-final-project/wisconsin/utils.py'>
```

In [42]:
```
df = pd.read_csv('breast-cancer-wisconsin.csv').set_index('id')
df.describe()
```

Out[42]:

	thickness	size_uniform	shape_uniform	adhesion	epithelial_size	\
count	683.000000	683.000000	683.000000	683.000000	683.000000	
mean	4.442167	3.150805	3.215227	2.830161	3.234261	
std	2.820761	3.065145	2.988581	2.864562	2.223085	
min	1.000000	1.000000	1.000000	1.000000	1.000000	
25%	2.000000	1.000000	1.000000	1.000000	2.000000	
50%	4.000000	1.000000	1.000000	1.000000	2.000000	
75%	6.000000	5.000000	5.000000	4.000000	4.000000	
max	10.000000	10.000000	10.000000	10.000000	10.000000	

	bare_nuclei	chromatin	normal_nuclei	mitoses	class
count	683.000000	683.000000	683.000000	683.000000	683.000000
mean	3.544656	3.445095	2.869693	1.603221	2.699854
std	3.643857	2.449697	3.052666	1.732674	0.954592
min	1.000000	1.000000	1.000000	1.000000	2.000000
25%	1.000000	2.000000	1.000000	1.000000	2.000000
50%	1.000000	3.000000	1.000000	1.000000	2.000000
75%	6.000000	5.000000	4.000000	1.000000	4.000000
max	10.000000	10.000000	10.000000	10.000000	4.000000

The "class" column is the y-variable (malignancy). Benign tumors have `class=2`, and malignant tumors have `class=4`. The other tumors correspond to measurements and counts of cell features represented as positive integers between 1 and 10. For example, `size_uniform` is a subjective measure of how uniform the cell sizes were in the biopsy slide, and `bare_nuclei` is an approximation to a count of the number of abnormal nuclei in the visible slice.

Note that in reading the CSV file I moved an additional column, "id", into the index of the DataFrame. This column listed codes for each observation which do not have much predictive power (except perhaps encoding information on when the sample was taken, which could allow a model to overfit to inconsistencies in the data collection method).

Note also that in the summary above, some of the columns had quite different means and variances from others. To fix this, I normalized (below) the X columns to have mean 0 and standard deviation 1. For convenience, I also renamed the benign category from 2 to 0 and the malignant category from 4 to 1. In the new DataFrame summary below, we can see that the columns now all have the same mean and standard deviations.

There were also 16 rows in the CSV with null values, indicated by question marks. To save time, I removed those rows with a simple regex in Emacs.

```
In [47]: # 2 is benign, 4 is malignant. We substitute 2 -> 0 and 4 -> 1.
         y = df['class'].replace(to_replace=2, value=0).astype('bool')
         X = df.drop('class', axis=1)

         # Mean-center and normalize
         for col in X:
             X[col] = (X[col] - X[col].mean()) / X[col].std()

In [48]: X.describe()

Out[48]:           thickness  size_uniform  shape_uniform      adhesion  \
         count  6.830000e+02  6.830000e+02   6.830000e+02  6.830000e+02
         mean   1.248391e-16  4.161304e-17  -6.241957e-17  6.241957e-17
         std    1.000000e+00  1.000000e+00   1.000000e+00  1.000000e+00
         min   -1.220297e+00 -7.016978e-01  -7.412304e-01 -6.388973e-01
         25%   -8.657829e-01 -7.016978e-01  -7.412304e-01 -6.388973e-01
         50%   -1.567545e-01 -7.016978e-01  -7.412304e-01 -6.388973e-01
         75%    5.522740e-01  6.032977e-01   5.971975e-01  4.083832e-01
         max    1.970331e+00  2.234542e+00   2.270232e+00  2.502944e+00

                epithelial_size   bare_nuclei     chromatin  normal_nuclei  \
         count     6.830000e+02  6.830000e+02  6.830000e+02   6.830000e+02
         mean     -2.080652e-17 -2.080652e-17  5.201631e-17   4.161304e-17
         std       1.000000e+00  1.000000e+00  1.000000e+00   1.000000e+00
         min      -1.005027e+00 -6.983413e-01 -9.981216e-01  -6.124785e-01
         25%      -5.552016e-01 -6.983413e-01 -5.899078e-01  -6.124785e-01
         50%      -5.552016e-01 -6.983413e-01 -1.816940e-01  -6.124785e-01
         75%       3.444489e-01  6.738310e-01  6.347336e-01   3.702689e-01
         max       3.043400e+00  1.771569e+00  2.675803e+00   2.335764e+00

                      mitoses
```

```
count  6.830000e+02
mean   9.362935e-17
std    1.000000e+00
min   -3.481446e-01
25%   -3.481446e-01
50%   -3.481446e-01
75%   -3.481446e-01
max    4.846139e+00
```

2 Machinery for model selection

To avoid re-writing the same code for each model selection run, I wrote some utility functions to compute several goodness-of-fit metrics in a model-agnostic way, as well as a model-agnostic cross-validation loop. In my `utils` module (code given at the end of this document), I define the following functions:

- `cross_validate(X, y, model, params=None, num_splits=5)`: Computes ROC curves for a given model and set of parameters over a given number of cross-validation folds. The parameter `model` is a function which takes as input a tuple `(split, param1, param2, ...)`, where `split` is a CV-split given inside the main loop and the `param1`, etc are given in the tuple `params`. For example, here is a model function for KNN:

  ```
  def knn(split, nbors):
      from sklearn.neighbors import KNeighborsClassifier
      X_train, X_val, y_train, y_val = split
      clf = KNeighborsClassifier(n_neighbors=nbors)
      clf.fit(X_train, y_train)
      return clf.predict_proba(X_val)[:,1]
  ```

  To change `nbors`, we would pass `params=(nbors,)` to `cross_validate`.

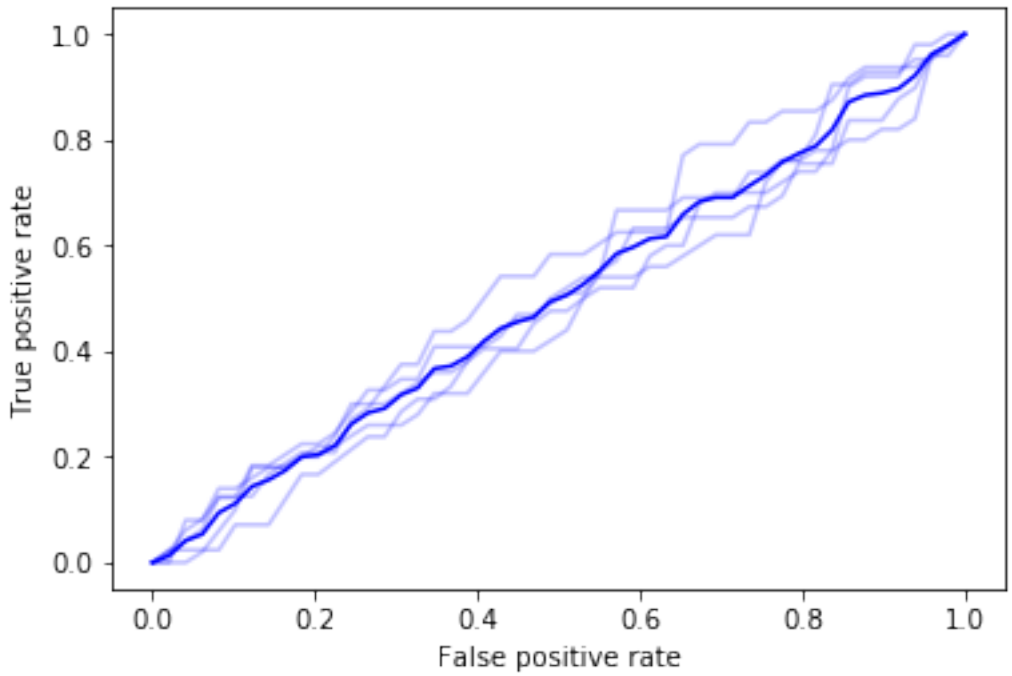
- `compute_roc(split, y_probs)`: Computes an ROC curve for a single train/test split given a predicted probability vector for the y-val observations. Does linear interpolation with 50 points to allow separate ROC curves to be averaged together.

- `def plot_roc_curves(curves, color='b', mean_only=False, show=True, label='')`: Takes curves computed by `compute_roc` and plots them on the same graph. Also averages curves and overlays the mean curve. The keyword arguments were added as the project progressed so that I could draw different types of graphs.

- Not listed: `auc` and `confusion_matrix` functions.

To verify that my model selection functions were implemented correctly, I tested them on a random model which simply predicts a uniform-random y-probability vector. As expected, I got a diagonal line in the ROC graph and an AUC (area under the curve) of extremely close to 0.5:

```
In [184]: curves = utils.cross_validate(X, y, models.random)
          utils.plot_roc_curves(curves)
          print(f'AUC for the random model: {utils.auc(curves)}')
```



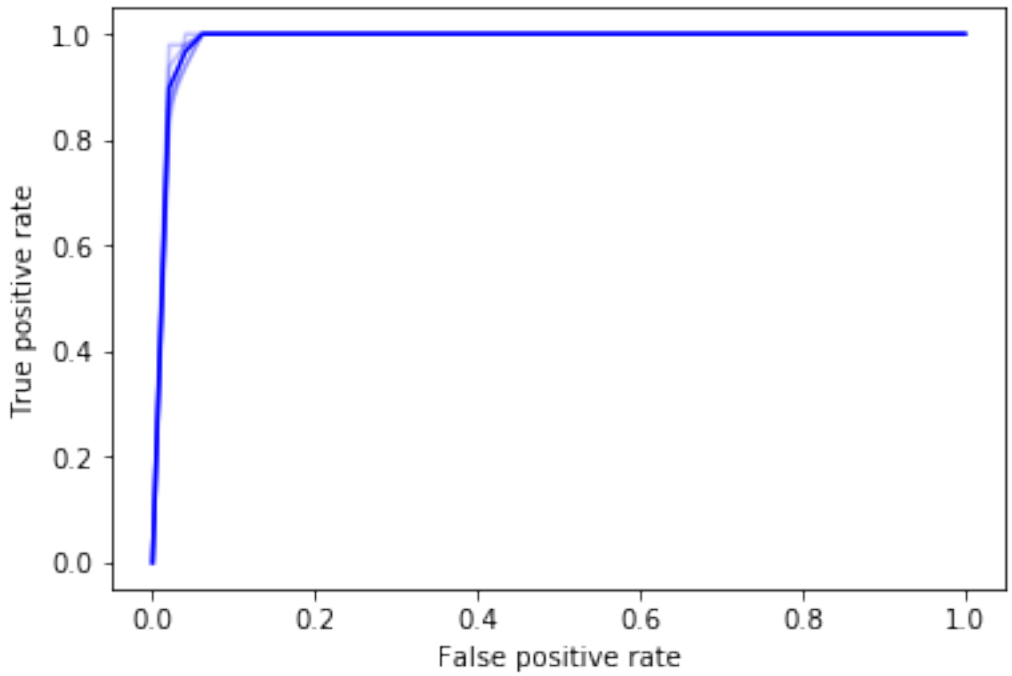

```
AUC for the random model: 0.5002148299319727
```

3 Models

3.1 Logistic model

As a baseline, I trained a logistic model (arguably the simplest classification model, since it simply fits a linear separation and then maps from feature-space to probability-space). The model performed surprisingly well, achieving an AUC of almost 0.976:

```
In [182]: lg_curves = utils.cross_validate(X, y, models.logistic)
          utils.plot_roc_curves(lg_curves)
          print(f'AUC for the logistic model: {utils.auc(curves)}')
```



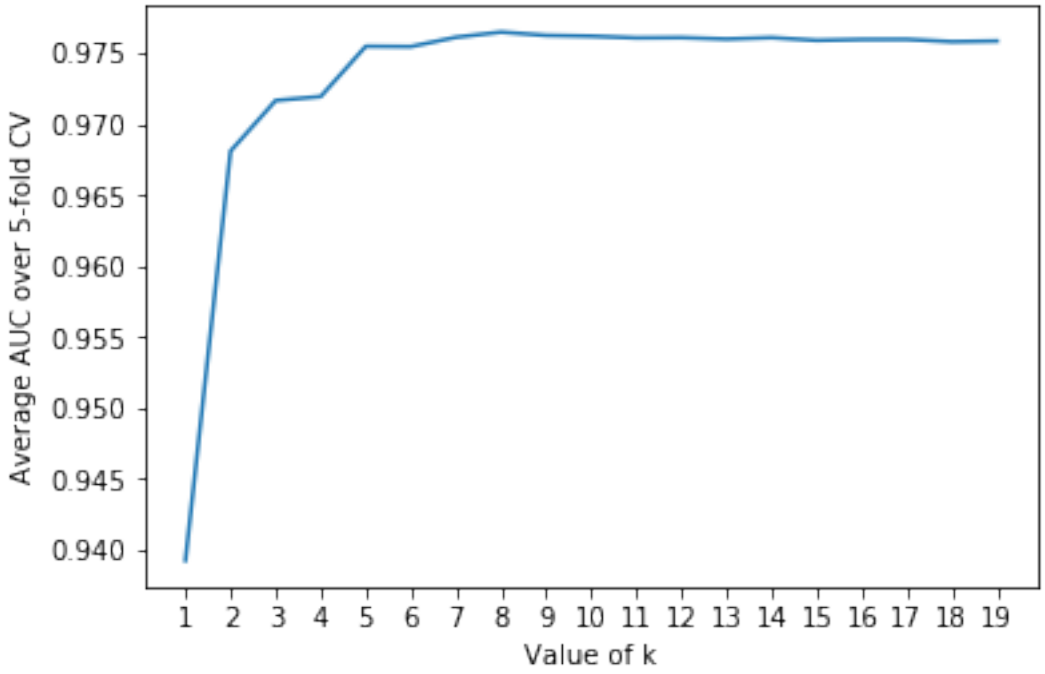

```
AUC for the logistic model: 0.975840459067981
```

3.2 *k*-Nearest-Neighbors

For kNN, it is necessary to set the hyperparameter k. I tested k values from 1 to 20 with 5-fold CV and found that $k = 8$ is optimal with respect to the AUC metric. The results are given in the plot below:

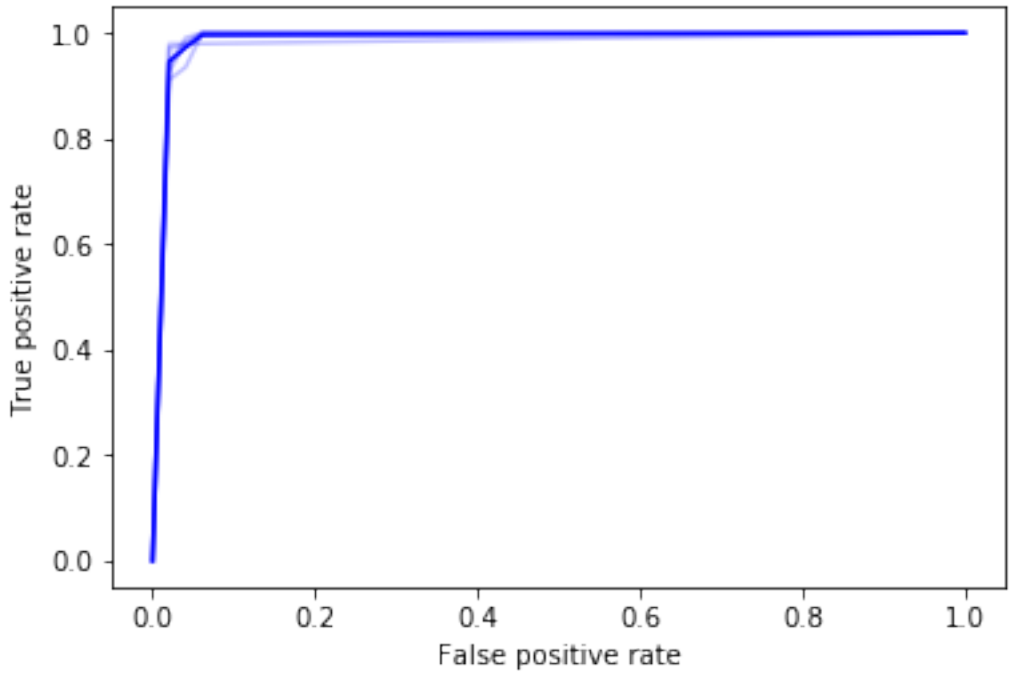
```
In [181]: AUCs = [0] * 20
          for nbors in range(1,20):
              curves = utils.cross_validate(X, y, models.knn, params=(nbors,))
              AUCs[nbors] = utils.auc(curves)

          plt.plot(range(1,20), AUCs[1:20])
          plt.xticks(range(1,20))
          plt.xlabel('Value of k')
          plt.ylabel('Average AUC over 5-fold CV')
          plt.show()
```



Perhaps surprisingly, the best kNN model didn't beat the logistic model! In fact, it got the exact same AUC. Here's an ROC plot for kNN with $k = 8$:

```
In [180]: knn_curves = utils.cross_validate(X, y, models.knn, params=(8,))
          utils.plot_roc_curves(knn_curves)
          print(f'Best AUC for knn: {utils.auc(curves)}')
```



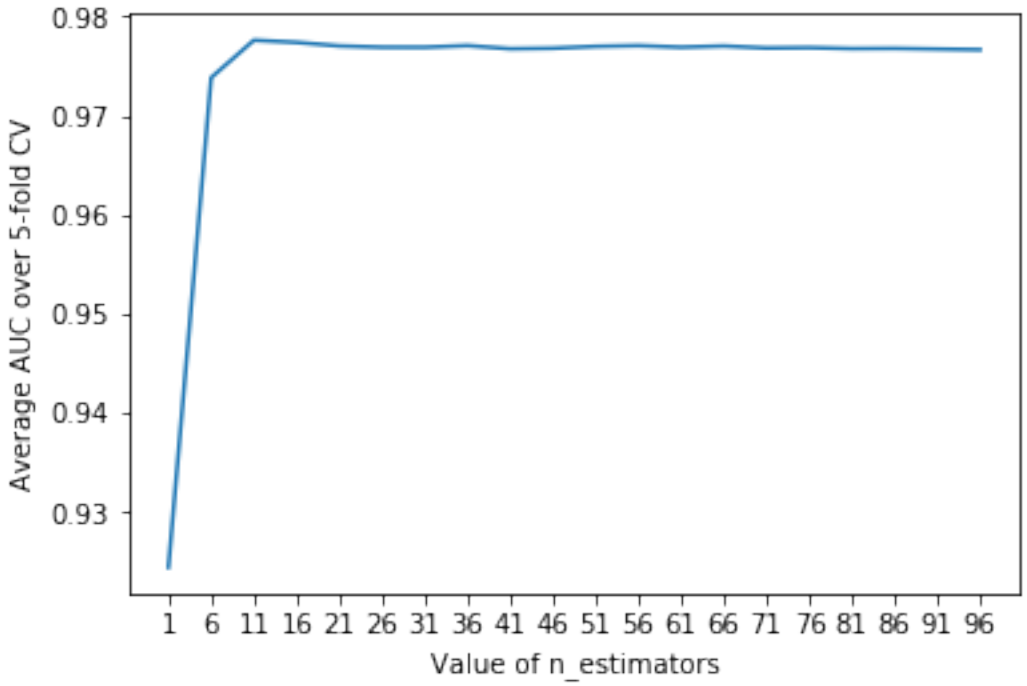
```
Best AUC for knn: 0.975840459067981
```

3.3 Random Forest

Random Forest is an ensemble method which fits a large number of weak, but expressive, classifiers, allowing it to handle a lot of the bias-variance tradeoff issue without requiring hyperparameter tuning. However, there are still some tunable parameters. For the purposes of this project I chose to tune `n_estimators`, the number of random trees in the ensemble, as the only hyperparameter. I tested it in increments of 5 from 1 to 100, and found that of those values 11 is optimal (as illustrated in the plot below). It's interesting that the graph of AUC vs. complexity for Random Forest plateaus much more quickly than the graph for kNN. A question for future work: is this because Random Forest is internally dealing with some of the bias-variance issues, or just a consequence of the random seed or the step size?

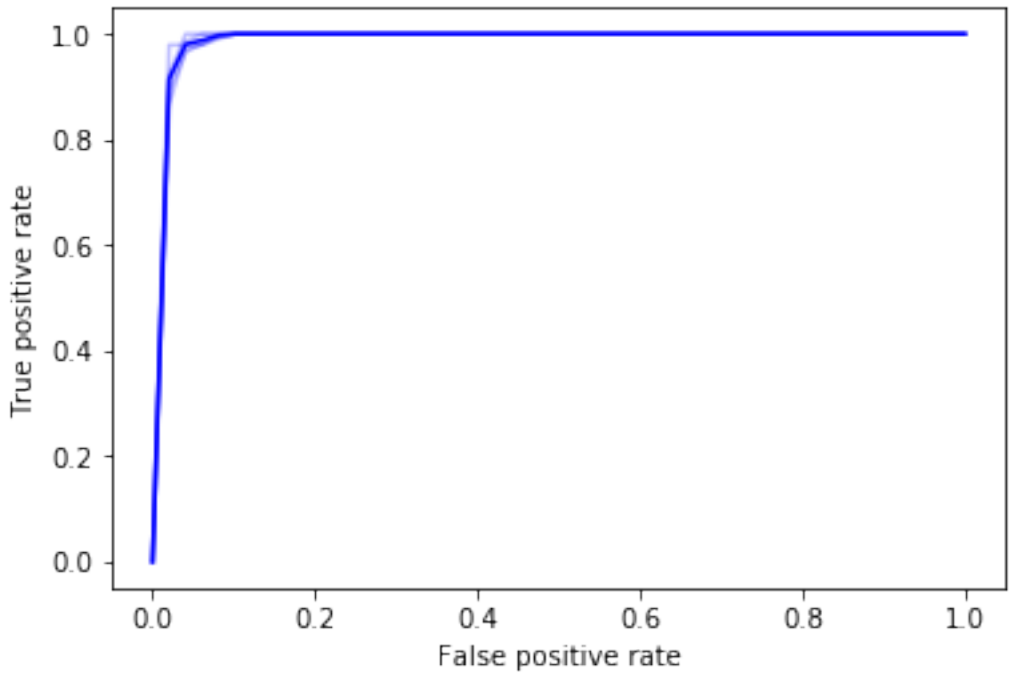
```
In [185]: params = range(1,100,5)
          AUCs = [0] * len(params)
          for i,n_estimators in enumerate(params):
              curves = utils.cross_validate(X, y, models.rf, params=(n_estimators,))
              AUCs[i] = utils.auc(curves)

          plt.plot(params, AUCs)
          plt.xticks(params)
          plt.xlabel('Value of n_estimators')
          plt.ylabel('Average AUC over 5-fold CV')
          plt.show()
```



As expected, Random Forest beats the "dumber" models I fit above, but only just *barely*: the AUC is about 0.9766, less than 0.002 higher than the previous models. This could easily be the result of noise in the splits or the small sample size. In future work, I would like to compute confidence intervals for the mean ROC curve to be able to reason more formally about the significance of AUC differences. The plot below shows the mean ROC for the best Random Forest model, with 11 estimators:

```
In [186]: rf_curves = utils.cross_validate(X, y, models.rf, params=(11,))
          utils.plot_roc_curves(rf_curves)
          print(f'Best AUC for random forest: {utils.auc(curves)}')
```




```
Best AUC for random forest: 0.9766513989541394
```

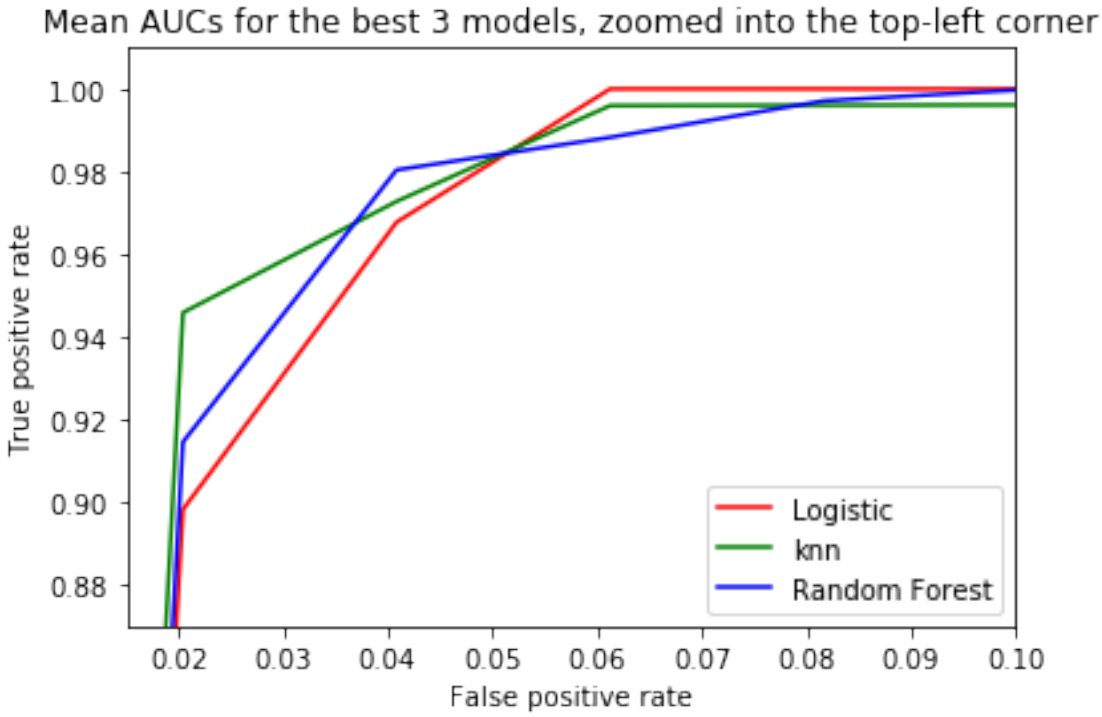
3.4 Comparison

To better visualize the differences between the models, I zoomed in on the top left corner of the ROC plots for each of the three model types. In the plot below, the mean ROC for the best variant of each model is shown:

```
In [188]: for curves, color,label in zip(
              [lg_curves, knn_curves, rf_curves],
              ['r','g','b'],
```

```
            ['Logistic', 'knn', 'Random Forest']):
            utils.plot_roc_curves(curves, color=color, mean_only=True, show=False, label=label

    plt.xlim(0.015, 0.1)
    plt.ylim(0.87,1.01)
    plt.legend(loc='lower right')
    plt.title('Mean AUCs for the best 3 models, zoomed into the top-left corner')
    plt.show()
```



Interestingly, the logistic model actually performs best for extremely low false-positive thresholds, as one might require in a medical diagnosis setting. Again, future work could add confidence intervals to make more rigorous statements about ROC differences between models.

```
In [200]: for model, params, label in zip(
              [models.logistic, models.knn, models.rf],
              [None, (8,), (11,)],
              ['Logistic', 'knn', 'Random Forest']
          ):
              print(f'Confusion matrix for {label}:
\n{utils.confusion_matrix(X, y, model, params=params)}')
```

I also computed confusion matrices across a single fold for each of the three models:

```
Confusion matrix for Logistic:
[[84  3]
 [ 3 47]]
Confusion matrix for knn:
```

```
[[84  3]
 [ 3 47]]
Confusion matrix for Random Forest:
[[84  3]
 [ 2 48]]
```

The confusion matrices for Logistic and knn were identical, which is consistent with their AUCs being identical as well (over 5 splits instead of 1). Random Forest improves slightly on the positive examples (malignant tumors), but does not improve on the previous models for negative examples (benign tumors).

4 Conclusion and future work

Unfortunately, the dataset I chose for my project turned out to be quite linearly separable, which made it difficult to differentiate more sophisticated methods from workhorses such as Logistic Regression and k-Nearest-Neighbors. On the other hand, the success of those simpler methods showed how on "real-world" data, simple methods are often good enough and should always be tried first before moving on to harder techniques.

All three models did extremely well with respect to both the AUC and confusion matrix metrics. It's likely that the couple of examples which were not correctly classified were simply outliers, and that we could expect to get similar performance from the models on real-world data. However, if I had to pick a single one of the models, based on the experimentation I've done so far I would go with the logistic regression model, as it did best in the extremely low false-positive-rate regime. In a medical diagnosis application, this regime would be critical for determining whether the model could be used in practice.

In future work on this dataset, as discussed above, I would like to get rigorous confidence intervals on the ROC curves, which would enable me to make stronger statements about the relative performances of the three models in various false positive regimes. I would also like to tackle harder datasets in the future to hone my skills using sophisticated algorithms such as Random Forest, Boosting, and Neural Nets. All things considered, however, I feel I learned a lot working through this easier dataset, and I will definitely be able to re-use the framework I designed for model selection in future projects.

5 Code: `models.py`

```
def random(split):
    import numpy as np
    X_train, X_val, y_train, y_val = split
    np.random.seed(0)
    return np.random.rand(len(y_val))

def knn(split, nbors):
    from sklearn.neighbors import KNeighborsClassifier
    X_train, X_val, y_train, y_val = split
    clf = KNeighborsClassifier(n_neighbors=nbors)
    clf.fit(X_train, y_train)
    return clf.predict_proba(X_val)[:,1]
```

```
def logistic(split):
    from sklearn.linear_model import LogisticRegression
    X_train, X_val, y_train, y_val = split
    clf = LogisticRegression(random_state=0, solver='lbfgs')
    clf.fit(X_train, y_train)
    return clf.predict_proba(X_val)[:,1]

def rf(split, n_estimators):
    from sklearn.ensemble import RandomForestClassifier
    X_train, X_val, y_train, y_val = split
    clf = RandomForestClassifier(n_estimators=n_estimators, random_state=0)
    clf.fit(X_train, y_train)
    return clf.predict_proba(X_val)[:,1]
```

6 Code: `utils.py`

```
import matplotlib.pyplot as plt
import numpy as np

# Compute the (interpolated) ROC of a list of probabilities y_probs
def compute_roc(split, y_probs):
    from sklearn.metrics import roc_curve
    X_train, X_val, y_train, y_val = split

    fpr_axis = np.linspace(0, 1)
    fpr, tpr, _ = roc_curve(y_val, y_probs, pos_label = 1)
    curve = np.interp(fpr_axis, fpr, tpr)
    curve[0] = 0.0
    return curve

# Plot a mean ROC curve and individual curves for the arrays fprs, tprs
def plot_roc_curves(curves, color='b', mean_only=False, show=True, label=''):
    mean_curve = np.array(curves).mean(axis=0)
    plt.plot(np.linspace(0,1), mean_curve, color,label=label)
    if mean_only == False:
        for curve in curves:
            plt.plot(np.linspace(0,1), curve, color, alpha=0.25)
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    if show:
        plt.show()

# Run CV splits on model
def cross_validate(X, y, model, params=None, num_splits=5):
    from sklearn.model_selection import KFold
    splits = KFold(n_splits=num_splits, shuffle=True, random_state=0).split(X)
```

```
    curves = []
    np.random.seed(0)

    for i, (train, val) in enumerate(splits):
    #    print(f'Split {i} of {num_splits}.')
        split = X.iloc[train], X.iloc[val], y.iloc[train], y.iloc[val]
        #  y_probs = models.knn_model(split, 400)
        if params is not None:
            y_probs = model(split, *params)
        else:
            y_probs = model(split)
        curve = compute_roc(split, y_probs)
        curves.append(curve)

    return curves

def confusion_matrix(X, y, model, params=None, num_splits=5):
    from sklearn.model_selection import KFold
    splits = KFold(n_splits=num_splits, shuffle=True, random_state=0).split(X)
    np.random.seed(0)

    for i, (train, val) in enumerate(splits):
        split = X.iloc[train], X.iloc[val], y.iloc[train], y.iloc[val]
        if params is not None:
            y_probs = model(split, *params)
        else:
            y_probs = model(split)
        y_pred = np.rint(y_probs) # round probabilities to 0/1
        break
    from sklearn.metrics import confusion_matrix
    return confusion_matrix(y.iloc[val], y_pred)

def auc(curves):
    mean_curve = np.array(curves).mean(axis=0)
    return sum(mean_curve) / len(mean_curve)
```