BUSA3020-Week7

February 13, 2023

Week 7 Lecture & Computer Lab - Best Practices for Model Evaluation and Hyperparameter Tuning

Unit Convenor & Lecturer

George Milunovich
george.milunovich@mq.edu.au

References

1. Python Machine Learning 3rd Edition by Raschka & Mirjalili - Chapter 6
2. Various open-source material

Week 7 Learning Objectives

1. Streamlining Workflows with Pipelines
2. The Holdout Method and K-Fold Cross-Validation to Assess Model Performance
3. Debugging algorithms with learning and validation curves
4. Fine-tuning machine learning models via grid search
5. Looking at different performance evaluation metrics

- The confusion matrix
- Optimizing the precision and recall of a classification model
- Plotting a receiver operating characteristic
- The scoring metrics for multiclass classification

Streamlining Workflows with Pipelines

- `Pipeline` class in scikit-learn is a wrapper tool
  - A wrapper is a class whose purpose is to provide a different interface than the thing it wraps
    - It usually provides more functionality and interface to it
  - It allows us to combine transformers and estimators in one object
  - We can make predictions about new data in one step
  - E.g. we can standarize features, extract principal components and fit a logistic regression all in one step

- `make_pipeline` function takes an arbitrary number of scikit-learn transformers (objects which support `fit` and `transform` methods) followed by an estimator that implements `fit` and `predict` methods
- `fit` method of `Pipeline` will pass the data down a series of transformers via `fit` and `transform` calls, until it reaches the estimator object
  - The estimator will then be fitted to the transformed training data
- `predict` method of `Pipeline` will pass the data through the intermediate steps via `transform` calls
  - In the final step the estimator wil return a prediction on the transformed data

Loading the Breast Cancer Wisconsin dataset

Breast Cancer Wisconsin dataset - 569 examples of malignant or benign tumor cells - 1st column - unique ID number of patient - 2nd column - diagnosis: M = Malignant, B = Benign - Columns 2 - 31 represent 30 features computed from digitized images of the cell nuclei used to build a model to predict whether a tumor is benign or malignant

```
import pandas as pd
import numpy as np

# df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wi
# df.to_csv('data/wdbc.data', index=False, header = None)

# if the Breast Cancer dataset is temporarily unavailable from the
# UCI machine learning repository, un-comment the following line
# of code to load the dataset from a local path:

df = pd.read_csv('data/wdbc.data', header = None)

print(df.shape)
df.head(10)

df
```

[ ]:

Example: Combining transformers and estimators in a pipeline

1. Create X and y variables
2. Encode target with `LabelEncoder`
3. Split dataset into train & test (20%) datasets stratifying by y
4. Use `make_pipeline` to
   - Standardize data (`StandardScaler`),
   - Extract two principal components (`PCA`)
   - Logistic Regression (`LogisticRegression`)
5. Fit to train dataset and compute accuracy on training data

6. Make predictions using test data and compute accuracy on test data
7. Repeat steps 4 - 6 by doing step-by-step of
    - Scaling data
    - Extracting principal components
    - Fitting Logistic Regression

1. Create X and y variables

```
print(df.shape)

y = df.loc[:, 1].values
X = df.loc[:, 2:].values

print(y.shape)
print(X.shape)
y[:25]
```

[ ]:

2. Encode target with `LabelEncoder`

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y = le.fit_transform(y)

print(le.classes_)
y
```

[ ]:

3. Split dataset into train & test (20%) datasets stratifying by y

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, stratify=y, random_st

print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
```

[ ]:

4. Use `make_pipeline` to
    - Standardize data (`StandardScaler`),
    - Extract two principal components (`PCA`)
    - Logistic Regression (`LogisticRegression`)

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.pipeline import make_pipeline

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(random_state=1, solver='lbfgs'))
```

[ ]:

5. Fit to train dataset and compute accuracy on training data

```
pipe_lr.fit(X_train, y_train)

print(f'Test Accuracy: {pipe_lr.score(X_train, y_train):.3f}')
```

[ ]:

6. Make predictions using test data and compute accuracy on test data

```
y_pred = pipe_lr.predict(X_test)
y_pred

print(f'Test Accuracy: {pipe_lr.score(X_test, y_test):.3f}')

# y_pred = pipe_lr.predict(X_test[0:1, :])
# y_pred
```

[ ]:

7. Repeat steps 4 - 6 by doing step-by-step of
   - Scaling data
   - Extracting principal components
   - Fitting Logistic Regression

```
sc = StandardScaler()
X_train_scaled = sc.fit_transform(X_train)
X_test_scaled = sc.transform(X_test)



pca = PCA(n_components = 2)
X_train_scaled_pca = pca.fit_transform(X_train_scaled)
X_test_scaled_pca = pca.transform(X_test_scaled)

lr = LogisticRegression(random_state=1, solver='lbfgs')
lr.fit(X_train_scaled_pca, y_train)
print(f'Train Accuracy: {lr.score(X_train_scaled_pca, y_train):.3f}')

print(f'Test Accuracy: {lr.score(X_test_scaled_pca, y_test):.3f}')
```

[ ]:

The Holdout Method and K-Fold Cross Validation to Assess Model Performance

- Choosing the best possible model for forecasting relies on
  - Training the model on the training dataset
  - Testing how well the model generalises on test data
  - However…
    - Also need to **tune hyperparameters** and compare different settings to further improve performance on unseen data
      - A hyperparameter is a parameter whose value is used to control the learning process
      - By contrast, the values of other parameters (typically called weights) are derived via training
    - **Model selection**: Selection of optimal values of **tuning parameters** (hyperparameters)
    - Problem: If we reuse the same test dataset over and over again during **model selection**, it will become part of our training data and thus the model will be more likely to overfit
- Need to be able to estimate how well the model can generalize (perform on unseen data) even when tuning hyperparameters
  - Holdout Cross-Validation
  - K-Fold Cross-Validation

The Holdout Method

- Split data into three parts:

1. Training dataset - fit different models and same models with different hyperparameter values
2. Validation dataset - **Model Selection**: repeatedly evaluate the performance of the model after training using different hyperparameter values
   - Choose best hyperparameter value
3. Test dataset - unseen data used to estimate final model's ability to generalise to new data

- **Disadvantage**:
  - Performance estimate is sensitive to how we partition the training dataset into the training and validation subsets

K-fold Cross-Validation

- Split the **training set** into k folds without replacement
  - $k-1$ folds are used for model training
  - 1 fold is used for performance evaluation
- Repeat this procedure k times until all k folds are used for evaluation (and training)
  - Compute average performance across all k folds to obtain a performance estimate that is less sensitive to sub-partitioning of the training data
- **Once best hyperparameter values are found** retrain the model on the complete training set (in order to fit on a maximum number of observations)

  - Final performance accuracy is computed using the independent test dataset

Advantages of K-fold Cross-Validation
- Averaging model performance (accuracy) across different validation datasets results in a more **precise** measure of performance, i.e. it will have a lower variance - Each example (observation) is used for validation exactly once

In the image below E_i's represent Estimated Performance meaure (e.g accuracy)

- Typically we set $k = 10$
  - If we work with a **small dataset** increase k in order to be able to train the model on more data
    - If very small dataset use $k = n$. This is called Leave-one-out cross-validation (LOOCV)
  - If we work with a **larger dataset** we can decrease k, e.g. $k = 5$
  - If we have unequal class proportions do **stratified cross-validation**
    - Use `from sklearn.model_selection import StratifiedKFold`

Previously we computed training set accuracy to be 0.954 - Lets re-compute this accuracy using K-Fold Cross-Validation - Note: in this example we do not tune hyperparameters yet - We just want to see how to average validation accuracies

```
# ---------------- Stratified KFold -------------------------

from sklearn.model_selection import StratifiedKFold

print(y_train.shape, y_test.shape)

# print(type(y_train.values))

kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)


# for k, (train, test) in enumerate(kfold):
#     print(k, '\n', test)
    # print(k,'\n', y_train[test])
    # print(k,'\n', np.bincount(y_train[train]) /len(train) ,'\n', np.bincount(y_train[test])/l

scores = []

for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])   # use pipe_lr to endure that all transformati
    score = pipe_lr.score(X_train[test], y_train[test])
    scores.append(score)
    print(f'Fold: {k+1}, Acc: {score}')

print(f'\nCV accuracy: {np.mean(scores):.3f} +/- {np.std(scores):.3f}')
```

[ ]:

In practice we are likely to use `from sklearn.model_selection import cross_val_score` -

`cross_val_score` library implements a **k-fold cross-validation** scorer - Allows us to do **stratified k-fold cross-validation** with less code - We don't need to loop through folds using `for` - https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

```
from sklearn.model_selection import cross_val_score

scores_v2 = cross_val_score(estimator=pipe_lr, X=X_train, y=y_train, cv=10, n_jobs=1)

# print(scores_v2)

print(f'CV accuracy scores\n {scores_v2.reshape(-1,1)}')
print(f'CV accuracy: {np.mean(scores_v2):.3f} +/- {np.std(scores_v2):.3f}')
print(f'Correlation between scores and scores2: {np.corrcoef(scores, scores_v2)[0,1]}')
```

[ ]:

Fine-Tuning Hyperparameter via Grid Search

There are two types of parameters in Machine Learning - Weights which are learned (estimated) from the training data - Hyperparamters (tuning parameters) which are set by the investigator, e.g. C regularization parameter in Logistic Regression - Different values of hyperparameters result in different forecast accuracy - How can we choose best hyperparameter values? - Try a lot of different values - **Grid Search** - a popular hyperparameter optimization technique

Tuning hyperparameters via grid search

- Brute-force exhaustive search method
  - Systematically list a lot of (sometimes all) possible values for the solution and check which value provides best solution

- Specify a list of values for different hyperparameters
- Algorithm evaluates the model performance for each combination to obtain the **optimal combination** of hyperparameter values from the specified set
- `from sklearn.model_selection import GridSearchCV`
- It is also possible to use RandomizedSearchCV to randomly sample parameter combinations via randomized search
  - It has been shown that if we sample 60 parameter combinations we have 95% chance of obtaining solutions within 5% of the optimal performance
  - https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Example - Use `GridSearchCV` with a Support Vector Machine classifier - https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html - Optimize the following hyperparameters - Regularization parameters C - inverse of regularization strength - Kernel: linear and rbf - γ parameter for rbf kernel - Print parameters and accuracy of the best model - Export best model and print accuracy on the test set

---

1. Create a `pipeline` containing

- `StandardScaler`
- `SVC`

```
from sklearn.model_selection import GridSearchCV

from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),
                         SVC(random_state=1, probability=True))
```

[ ]:

---

2. Define a parameter range & parameter grid

```
param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]  # range of values for C and


param_grid = [{'svc__C': param_range,   # range of values for all parameters
               'svc__kernel': ['linear']},
              {'svc__C': param_range,
               'svc__gamma': param_range,
               'svc__kernel': ['rbf']}]
```

[ ]:

3.

- Initialize `GridSearchCV`
- Fit to data
- Print accuracy and optimized hyperparameters of best model

```
gs = GridSearchCV(estimator=pipe_svc,      # initialise gs object
                  param_grid=param_grid2,
                  scoring='accuracy',
                  refit=True,              # this will refit the best estimator to the whole da
                  cv=10,
                  n_jobs=-1)


gs = gs.fit(X_train, y_train)            # fit gs

print(gs.best_score_)

print(gs.best_params_)
```

[ ]:

---

4. Export best model

- Compute accuracy on test data
- Print predicted labels for test data

```
best_classifier = gs.best_estimator_                # copy best estimator

print(best_classifier)
print(f'Test accuracy: {best_classifier.score(X_test, y_test):.3f}')
print(best_classifier.predict(X_test))
```

[ ]:

Debugging Algorithms with Learning and Validation Curves

Two diagnostic tools to help improve the performance of a learning algorithm - Learning Curves - performance as a function of sample size - Validation Curves - performance as a function of model hyperparameters

Diagnosing Bias and Variance Problems with Learning Curves

If a model is too complex - has too many parameters - it is likely to overfit - By increasing the sample size we can reduce the extent of overfitting - In practice it may be very costly or impossible to collect more data - Plot model **training** and **validation** accuracy as functions of training dataset size - Detect whether the model suffers from high variance or high bias - Find out if more data can reduce these problems

Model in upper-left corner - Low training and cross-validation accuracy -> model underfits -> high bias - Possible solution - increase the number of parameters - Collect or construct additional features - Decrease the degree of regularization

Model in upper-right corner - Large variability between training and cross-validation accuracy -> model overfits -> high variance - Collect more training data - Reduce the complexity of the model (number of parameters/features) - e.g. via Feature Extraction/Selection - Increase the degree of regularization

In scikit-learn use `from sklearn.model_selection import learning_curve` - By default uses **stratified k-fold cross-validation** - https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.learning_curve.html - Lets implement this on our training dataset

[58]:
```
# import matplotlib.pyplot as plt

# from sklearn.model_selection import learning_curve
```

```
# pipe_lr = make_pipeline(StandardScaler(),
#                         LogisticRegression(penalty='l2',
#                                            random_state=1,
#                                            solver='lbfgs',
#                                            max_iter=10000)) # increase number␣
 ↪of iterations for optimization algorithm in order to avoid potential␣
 ↪problems with small datasets


# train_sizes, train_scores, test_scores = learning_curve(estimator=pipe_lr,
#                                X=X_train,
#                                y=y_train,
#                                train_sizes=np.linspace(0.1, 1.0, 11), #␣
 ↪train_sizes are set up here, split interval [0,1] into 11 evenly spaced␣
 ↪pieces
#                                cv=10,
#                                n_jobs=1)


# # print('train sizes', train_sizes)
# # print('train scores', train_scores.shape, train_scores)
# # print('test scores', test_scores.shape, test_scores)

# train_mean = np.mean(train_scores, axis=1)
# train_std = np.std(train_scores, axis=1)
# test_mean = np.mean(test_scores, axis=1)
# test_std = np.std(test_scores, axis=1)

# plt.plot(train_sizes, train_mean,
#          color='blue', marker='o',
#          markersize=5, label='Training accuracy')

# plt.fill_between(train_sizes,      # plot mean +- one standard deviation     ␣
 ↪
#                  train_mean + train_std,
#                  train_mean - train_std,
#                  alpha=0.15, color='blue')

# plt.plot(train_sizes, test_mean,
#          color='green', linestyle='--',
#          marker='s', markersize=5,
#          label='Validation accuracy')

# plt.fill_between(train_sizes,
#                  test_mean + test_std,
#                  test_mean - test_std,
#                  alpha=0.15, color='green')
```

```
# plt.grid()
# plt.xlabel('Number of training examples')
# plt.ylabel('Accuracy')
# plt.legend(loc='lower right')
# plt.ylim([0.8, 1.03])
# plt.tight_layout()
# # plt.savefig('images/06_05.png', dpi=300)
# plt.show()
```

Conclusion: - The model performs quite well for sample sizes of about 250 observations or greater

Addressing Over- and Under-Fitting with Validation Curves

- Validation Curves - performance as a function of model hyperparameters
  - `from sklearn.model_selection import validation_curve`
  - `validation_curve` employs **stratified k-fold cross-validation** to estimate the performance of the classifier
  - Need to specify the parameter we wish to evaluate in `validation_curve`
  - Also need to specify a parameter range that the evaluated parameter will take
  - https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.validation_curve.html

Example - Employ a validation curve on LogisticRegression - Vary C - the inverse of the regularization parameter - over the range `[0.001, 0.01, 0.1, 1.0, 10.0, 100.0]`

[57]:
```
# from sklearn.model_selection import validation_curve


# param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]

# train_scores, test_scores = validation_curve(
#                 estimator=pipe_lr,
#                 X=X_train,
#                 y=y_train,
#                 param_name='logisticregression__C',
#                 param_range=param_range,
#                 cv=10)

# train_mean = np.mean(train_scores, axis=1)
# train_std = np.std(train_scores, axis=1)
# test_mean = np.mean(test_scores, axis=1)
# test_std = np.std(test_scores, axis=1)

# plt.plot(param_range, train_mean,
#          color='blue', marker='o',
#          markersize=5, label='Training accuracy')
```

```
# plt.fill_between(param_range, train_mean + train_std,
#                  train_mean - train_std, alpha=0.15,
#                  color='blue')

# plt.plot(param_range, test_mean,
#          color='green', linestyle='--',
#          marker='s', markersize=5,
#          label='Validation accuracy')

# plt.fill_between(param_range,
#                  test_mean + test_std,
#                  test_mean - test_std,
#                  alpha=0.15, color='green')

# plt.grid()
# plt.xscale('log')
# plt.legend(loc='lower right')
# plt.xlabel('Parameter C')
# plt.ylabel('Accuracy')
# plt.ylim([0.8, 1.0])
# plt.tight_layout()
# # plt.savefig('images/06_06.png', dpi=300)
# plt.show()
```

- Small C -> regularization is too strong -> both training and validation accuracy is relatively low -> underfitting
- $C \in \{0.1, 1\}$ seems to be best in terms of both training and validation accuracy
- $C > 1$ -> regularization is too weak -> too many parameters -> training accuracy is good but validation accuracy decreases -> overfitting

---

Looking at Different Performance Evaluation Metrics

- So far we have relied on **prediction accuracy** as a measure of performance
- Now we'll look at several other measures of performance

The Confusion Matrix

This should be familiar from basic statistics

- The null hypothesis H_0 is usually something that we assume prior to test, e.g. innocent until proven guilty, healthy until tested positive for a desease, etc
  - Negative Result -> Accept H0: e.g. negative COVID test -> accept H0: no covid
  - Positive Result -> Reject H0: e.g. positive COVID test -> reject H0: no covid
- We can extend this as follows
  - False Positive (FP) -> False = Incorrect, Positive = Reject H0 -> Reject H0 when True = Type I Error, e.g. diagnosed with COVID when in reality not sick

  - False Negative (FN) -> False = Incorrect, Negative = Accept H0 -> Accept H0 when False = Type II Error, e.g. not diagnosed with COVID when in reality sick
- Use `from sklearn.metrics import confusion_matrix`
  - Need to specify true classes (target) and predicted classes (prediction of target)

Example

- Retrain `pipe_svc` on train dataset
- Produce predictions using `X_test` dataset
- Compute and plot confusion matrix
- Interpret the results

1.

- Retrain `pipe_svc` on train dataset
- Produce predictions using `X_test` dataset

```
from sklearn.metrics import confusion_matrix

import matplotlib.pyplot as plt

import seaborn as sns


pipe_svc.fit(X_train, y_train)

y_pred = pipe_svc.predict(X_test)
```

[ ]:

2.

- Compute and plot confusion matrix
- Interpret the results

```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)

# ---------- Plotting
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i, s=confmat[i, j], va='center', ha='center')


plt.xlabel('Predicted label')
plt.ylabel('True label')

plt.tight_layout()
```

```
#plt.savefig('images/06_09.png', dpi=300)
plt.show()
```

[ ]:

In our dataset there are classes: 0 = Benign tumor and 1 = Malignant tumor

H_0: Benign tumor (class 0)
H_1: Malignant tumor (class 1)

- Correctly Classified: 71 examples of class 0 and 40 examples of class 1
- Misclassified:
  - False Positive (false rejection of H_0) 1 example
  - False Negative (false acceptance of H_0) 2 examples

Optimizing the Precision and Recall of a Classification Model

- Prediction Error: $ERR = \frac{\text{Misclassified}}{N} = \frac{FP+FN}{FP+FN+TP+TN}$
- Accuracy: $ACC = \frac{\text{Correctly Classified}}{N} = \frac{TP+TN}{FP+FN+TP+TN} = 1 - ERR$

We select the best forecasting model based on some performance measure, such as accuracy above - However, in some circumstances we may want to choose a model according to some other performance measure, such as the TPR below

True Positive and False Positive Rates

- True Positive Rate = $TPR = \frac{TP}{P} = \frac{TP}{TP+FN}$
  - Probablity of correctly detecting effect
  - TPR is known as **power** in statistical testing
- False Positive Rate = $FPR = \frac{FP}{N} = \frac{FP}{FP+TN}$
  - Probability of incorrectly detecting effect
  - Known as the Probability of Type I Error in statistical testing
- More concerned with TPR - detecting malignant tumors
  - Want TPR as high as possible
  - Optimizing TPR therefore may be more important than choosing the model with the highest accuracy in this case
- It is also important to miminize FPR - not to concern patients unnecessarily

Plotting Confusion Matrix v.2

```
labels = [1,0]
confmat2 = confusion_matrix(y_true=y_test, y_pred=y_pred, labels = labels)

group_counts = ["{0:0.0f}".format(value) for value in confmat2.flatten()]
# group_percentages = ["{0:.2%}".format(value) for value in confmat2.flatten()/np.sum(confmat2)
group_names = ['True Pos', 'False Neg','False Pos', 'True Neg']
```

```
labels3 = [f"{v2}\n{v3}" for v2, v3 in zip(group_names, group_counts)]
# labels3 = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in zip(group_names,group_counts,group_percentag



labels3 = np.asarray(labels3).reshape(2,2)

fig, ax = plt.subplots()
sns.heatmap(confmat2, annot=labels3, fmt='',xticklabels=labels, yticklabels=labels, cmap='Blues

ax.set_xlabel('Predicted label')
ax.xaxis.set_label_position('top')
ax.set_ylabel('True label')

plt.tick_params(axis='both', which='major', labelsize=10, labelbottom = False, bottom=False, to

plt.show()
```

[ ]:

Precision (PRE) and Recall (REC)

- $REC = TPR = \frac{TP}{TP+FN} = 0.95$
  - Optimizing (choosing) a model based on REC will minimize the chance of not detecting a malignant tumor
    - This will however result in some predictions of malignant tumors in healthy patients (FP)
- $PRE = \frac{TP}{TP+FP} = 0.98$
  - PRE = 100% will result in having no False Positives
    - Likely to miss malignant tumors more frequently (FN)
    - But good for spam filtering -> don't want to classify real emails as spam
- F1 score is a combination of PRE and REC $F1 = 2\frac{PRE \times REC}{PRE+REC}$
- In scikit-learn these scoring matrics are already implemented
  - `from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score`
- If we want to use a scoring metric other than `accuracy` in `GridSearchCV` we can change the `scoring` parameter
  - https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

```
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

print(f'Accuracy: {accuracy_score(y_true=y_test, y_pred=y_pred):.3f}')
print(f'Precision: {precision_score(y_true=y_test, y_pred=y_pred):.3f}')
print(f'Recall: {recall_score(y_true=y_test, y_pred=y_pred):.3f}')
print(f'F1: {f1_score(y_true=y_test, y_pred=y_pred):.3f}')
```

[ ]:

Appendix 1: The Scoring Metrics for Multiclass Classification

- Binary scoring methods can be extended to multiclass problems via one-vs-all (OvA) classification
- See textbook for details

Appendix 2: Dealing with Class Imbalance

- Class Imbalance occures quite often in real world
  - Examples from one class or multiple classes are over-represented in a dataset
  - Lets say class1 = 90 examples, class2 = 10 examples -> if we just predict class1 for all examples -> 90% accuracy
    - If an ML model returns 90% accuracy -> it hasn't really learned anything useful from the features
  - In this case accuracy is not the most useful measure of performance
    - If want to identify the majority of patients with malignant cancer to recommend an additional screening -> use REC (recall)
    - If screening for spam email -> use PRE (precision) (minimise the prob of classifying real emails as spam)
  - E.g. fraud detection, loan defaults, etc

Potential solutions - Assign a larger penalty to wrong predictions on the minority class - Set `class_weight='balanced'` parameter - Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)). - Upsampling of the minority class - `resample` in scikit-learn - Repeatedly draw new samples of the minority class from the dataset with replacement - Can also Downsample the majority class - Generate new (synthetic) examples of the minority class - Too technical - Synthetic Minority Oversampling Technique (SMOTE)

- Lets create an imbalanced dataset -> 357 begnin tumors (class 0) and 40 malignant (class 1)

[71]:
```
# X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
# y_imb = np.hstack((y[y == 0], y[y == 1][:40]))

# print(y_imb.shape)
# print(np.bincount(y_imb))
# print(np.bincount(y_imb)/len(y_imb))
```

- So if we predict y = 0 for all samples -> about 90% accuracy

[70]:
```
# from sklearn.utils import resample

# print('Number of class 1 examples before:', X_imb[y_imb == 1].shape[0])

# X_upsampled, y_upsampled = resample(X_imb[y_imb == 1],
#                                     y_imb[y_imb == 1],
#                                     replace=True,
```

```
#                                       n_samples=X_imb[y_imb == 0].shape[0],
#                                       random_state=123)

# print('Number of class 1 examples after:', X_upsampled.shape[0])
# pd.DataFrame(X_upsampled)
```

[69]:
```
# X_bal = np.vstack((X[y == 0], X_upsampled))
# y_bal = np.hstack((y[y == 0], y_upsampled))
```

[68]:
```
# print(y_bal.shape)
# print(np.bincount(y_bal))
# print(np.bincount(y_bal)/len(y_bal))
```

Appendix 3: Receiver Operating Characteristic (ROC)

Receiver Operating Characteristic (ROC) graphs are useful to select models for classification based on their performance with respect to FPR and TPR
- https://en.wikipedia.org/wiki/Receiver_operating_characteristic
- Plot TPR (power) vs FPR (Type I Error) - Shift the decision threshold of the classifier
- Diagonal interpreted as random guessing
- Classification models below the diagonal are worse than random guessing
- A perfect classifier is in the top-lect corner with a TPR=1 and FPR=0
- ROC area under the curve (ROC AUC) can be computed to characterize performance - ROC AUC = 1 -> perfect classifier - ROC AUC = 0.5 -> random guessing

Example

- Implement ROC on Breast Cancer Data
- Vary X from all features to only 2 features, see how it changes FPR and TPR

[67]:
```
# from sklearn.metrics import roc_curve, auc
# from numpy import interp

# pipe_lr = make_pipeline(StandardScaler(),
#                         PCA(n_components=2),
#                         LogisticRegression(penalty='l2',
#                                            random_state=1,
#                                            solver='lbfgs',
#                                            C=100.0))

# # X_train2 = X_train[:, [4, 14]]
# X_train2 = X_train.copy()


# cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))

# fig = plt.figure(figsize=(7, 5))
```

```
# mean_tpr = 0.0
# mean_fpr = np.linspace(0, 1, 100)
# all_tpr = []

# for i, (train, test) in enumerate(cv):
#     probas = pipe_lr.fit(X_train2[train],
#                          y_train[train]).predict_proba(X_train2[test])

#     fpr, tpr, thresholds = roc_curve(y_train[test],
#                                      probas[:, 1],
#                                      pos_label=1)
#     mean_tpr += interp(mean_fpr, fpr, tpr)
#     mean_tpr[0] = 0.0
#     roc_auc = auc(fpr, tpr)
#     plt.plot(fpr,
#              tpr,
#              label='ROC fold %d (area = %0.2f)'
#                    % (i+1, roc_auc))

# plt.plot([0, 1],
#          [0, 1],
#          linestyle='--',
#          color=(0.6, 0.6, 0.6),
#          label='Random guessing')

# mean_tpr /= len(cv)
# mean_tpr[-1] = 1.0
# mean_auc = auc(mean_fpr, mean_tpr)
# plt.plot(mean_fpr, mean_tpr, 'k--',
#          label='Mean ROC (area = %0.2f)' % mean_auc, lw=2)
# plt.plot([0, 0, 1],
#          [0, 1, 1],
#          linestyle=':',
#          color='black',
#          label='Perfect performance')

# plt.xlim([-0.05, 1.05])
# plt.ylim([-0.05, 1.05])
# plt.xlabel('False positive rate')
# plt.ylabel('True positive rate')
# plt.legend(loc="lower right")

# plt.tight_layout()
# # plt.savefig('images/06_10.png', dpi=300)
# plt.show()
```

[ ]: