[Handling Imbalanced Datasets] [cheatsheet]

1. Importing Required Libraries

- Import NumPy: `import numpy as np`
- Import Pandas: `import pandas as pd`
- Import Matplotlib: `import matplotlib.pyplot as plt`
- Import Seaborn: `import seaborn as sns`
- Import Scikit-learn: `from sklearn import *`
- Import Imbalanced-learn: `from imblearn import *`

2. Loading and Exploring Data

- Load data from a CSV file: `data = pd.read_csv('dataset.csv')`
- Load data from an Excel file: `data = pd.read_excel('dataset.xlsx')`
- Load data from a SQL database: `data = pd.read_sql('SELECT * FROM table', connection)`
- Explore the shape of the data: `print(data.shape)`
- View the first few rows of the data: `print(data.head())`
- Check the data types of columns: `print(data.dtypes)`
- Check for missing values: `print(data.isnull().sum())`
- Get summary statistics of numerical columns: `print(data.describe())`
- Count the occurrences of each class: `print(data['target'].value_counts())`
- Calculate the class distribution percentages: `print(data['target'].value_counts(normalize=True))`

3. Visualizing Class Imbalance

- Create a bar plot of class distribution: `data['target'].value_counts().plot(kind='bar')`
- Create a pie chart of class distribution: `data['target'].value_counts().plot(kind='pie')`
- Create a histogram of a feature colored by class: `sns.histplot(data, x='feature', hue='target')`
- Create a boxplot of a feature colored by class: `sns.boxplot(x='target', y='feature', data=data)`
- Create a scatter plot of two features colored by class: `sns.scatterplot(x='feature1', y='feature2', hue='target', data=data)`

By: Waleed Mousa

- Create a pair plot of features colored by class: `sns.pairplot(data, hue='target')`

4. Resampling Techniques

- Perform random undersampling: `undersampled_data = under_sampling.RandomUnderSampler().fit_resample(X, y)`
- Perform random oversampling: `oversampled_data = over_sampling.RandomOverSampler().fit_resample(X, y)`
- Perform undersampling with Tomek links: `undersampled_data = under_sampling.TomekLinks().fit_resample(X, y)`
- Perform oversampling with SMOTE: `oversampled_data = over_sampling.SMOTE().fit_resample(X, y)`
- Perform oversampling with ADASYN: `oversampled_data = over_sampling.ADASYN().fit_resample(X, y)`
- Perform undersampling with cluster centroids: `undersampled_data = under_sampling.ClusterCentroids().fit_resample(X, y)`
- Perform oversampling with borderline SMOTE: `oversampled_data = over_sampling.BorderlineSMOTE().fit_resample(X, y)`
- Perform oversampling with SVMSMOTE: `oversampled_data = over_sampling.SVMSMOTE().fit_resample(X, y)`
- Perform undersampling with instance hardness threshold: `undersampled_data = under_sampling.InstanceHardnessThreshold().fit_resample(X, y)`
- Perform combination of over- and undersampling with SMOTEENN: `resampled_data = combine.SMOTEENN().fit_resample(X, y)`
- Perform combination of over- and undersampling with SMOTETomek: `resampled_data = combine.SMOTETomek().fit_resample(X, y)`

5. Cost-Sensitive Learning

- Create a cost matrix: `cost_matrix = [[0, 1], [5, 0]]`
- Train a decision tree classifier with class weights: `clf = tree.DecisionTreeClassifier(class_weight={0: 1, 1: 5})`
- Train a random forest classifier with class weights: `clf = ensemble.RandomForestClassifier(class_weight={0: 1, 1: 5})`
- Train a logistic regression classifier with class weights: `clf = linear_model.LogisticRegression(class_weight={0: 1, 1: 5})`
- Train a support vector machine with class weights: `clf = svm.SVC(class_weight={0: 1, 1: 5})`

- Train a gradient boosting classifier with class weights: clf = ensemble.GradientBoostingClassifier(class_weight={0: 1, 1: 5})
- Train a weighted random forest classifier: clf = ensemble.RandomForestClassifier(n_estimators=100, random_state=42, class_weight='balanced')

6. Ensemble Techniques

- Train a balanced bagging classifier: clf = ensemble.BalancedBaggingClassifier(base_estimator=tree.DecisionTreeClassifier(), sampling_strategy='auto')
- Train a balanced random forest classifier: clf = ensemble.BalancedRandomForestClassifier(n_estimators=100, random_state=42)
- Train an easy ensemble classifier: clf = ensemble.EasyEnsembleClassifier(n_estimators=10, random_state=42)
- Train a RUSBoost classifier: clf = ensemble.RUSBoostClassifier(n_estimators=50, random_state=42)

7. Threshold Moving

- Predict probabilities using a trained classifier: probabilities = clf.predict_proba(X_test)
- Adjust the decision threshold: y_pred = (probabilities[:, 1] >= 0.3).astype(int)
- Plot the ROC curve: metrics.plot_roc_curve(clf, X_test, y_test)
- Plot the precision-recall curve: metrics.plot_precision_recall_curve(clf, X_test, y_test)
- Find the optimal threshold using Youden's J statistic: fpr, tpr, thresholds = metrics.roc_curve(y_test, probabilities[:, 1])
- Find the optimal threshold using F1 score: precision, recall, thresholds = metrics.precision_recall_curve(y_test, probabilities[:, 1])

8. Evaluation Metrics

- Calculate accuracy score: accuracy = metrics.accuracy_score(y_test, y_pred)
- Calculate balanced accuracy score: balanced_accuracy = metrics.balanced_accuracy_score(y_test, y_pred)
- Calculate precision score: precision = metrics.precision_score(y_test, y_pred)

- Calculate recall score: `recall = metrics.recall_score(y_test, y_pred)`
- Calculate F1 score: `f1 = metrics.f1_score(y_test, y_pred)`
- Calculate area under the ROC curve: `roc_auc = metrics.roc_auc_score(y_test, y_pred)`
- Calculate average precision score: `ap = metrics.average_precision_score(y_test, y_pred)`
- Generate a classification report: `report = metrics.classification_report(y_test, y_pred)`
- Generate a confusion matrix: `cm = metrics.confusion_matrix(y_test, y_pred)`
- Plot a confusion matrix: `sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')`

9. Feature Selection and Dimensionality Reduction

- Perform univariate feature selection with ANOVA F-test: `selector = feature_selection.SelectKBest(score_func=feature_selection.f_classif, k=10)`
- Perform recursive feature elimination: `selector = feature_selection.RFE(estimator=svm.SVC(), n_features_to_select=10)`
- Perform recursive feature elimination with cross-validation: `selector = feature_selection.RFECV(estimator=svm.SVC(), step=1, cv=5)`
- Perform principal component analysis (PCA): `pca = decomposition.PCA(n_components=10)`
- Perform linear discriminant analysis (LDA): `lda = discriminant_analysis.LinearDiscriminantAnalysis(n_components=2)`
- Perform t-distributed stochastic neighbor embedding (t-SNE): `tsne = manifold.TSNE(n_components=2)`

10. Model Interpretation and Explainability

- Visualize feature importances of a decision tree: `tree.plot_tree(clf)`
- Visualize feature importances of a random forest: `ensemble.plot_feature_importances(clf)`
- Plot permutation feature importance: `inspection.permutation_importance(clf, X_test, y_test)`
- Plot partial dependence: `inspection.plot_partial_dependence(clf, X_test, features=['feature1', 'feature2'])`
- Plot individual conditional expectation (ICE): `inspection.plot_partial_dependence(clf, X_test, features=['feature1', 'feature2'], kind='individual')`

- Plot SHAP values: `shap.summary_plot(shap_values, X_test)`
- Plot LIME explanations: `explainer = lime.lime_tabular.LimeTabularExplainer(X_train, feature_names=feature_names, class_names=class_names, discretize_continuous=True)`

11. Hyperparameter Tuning

- Perform grid search cross-validation: `param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}; clf = model_selection.GridSearchCV(svm.SVC(), param_grid, cv=5)`
- Perform random search cross-validation: `param_dist = {'C': uniform(0.1, 10), 'kernel': ['linear', 'rbf']}; clf = model_selection.RandomizedSearchCV(svm.SVC(), param_dist, n_iter=10, cv=5)`
- Perform Bayesian optimization: `optimizer = skopt.BayesSearchCV(svm.SVC(), {'C': (0.1, 10, 'log-uniform'), 'kernel': ['linear', 'rbf']}, n_iter=10, cv=5)`
- Perform hyperparameter tuning with imbalanced-learn: `param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}; clf = model_selection.GridSearchCV(svm.SVC(class_weight='balanced'), param_grid, cv=5, scoring='f1')`

12. Model Selection and Evaluation

- Split data into train and test sets: `X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.2)`
- Perform stratified k-fold cross-validation: `scores = model_selection.cross_val_score(clf, X, y, cv=model_selection.StratifiedKFold(n_splits=5))`
- Perform repeated stratified k-fold cross-validation: `scores = model_selection.cross_val_score(clf, X, y, cv=model_selection.RepeatedStratifiedKFold(n_splits=5, n_repeats=3))`
- Perform nested cross-validation: `scores = model_selection.cross_val_score(model_selection.GridSearchCV(svm.SVC(), param_grid, cv=5), X, y, cv=5)`
- Perform model evaluation with imbalanced-learn: `scores = model_selection.cross_val_score(svm.SVC(class_weight='balanced'), X, y, cv=5, scoring='f1')`

13. Handling Multi-Class Imbalance

- Perform one-vs-rest (OvR) classification: `clf = multiclass.OneVsRestClassifier(svm.SVC())`
- Perform one-vs-one (OvO) classification: `clf = multiclass.OneVsOneClassifier(svm.SVC())`
- Perform multi-class oversampling with SMOTE: `oversampled_data = over_sampling.SMOTENC(categorical_features=[0, 1]).fit_resample(X, y)`
- Perform multi-class undersampling with TomekLinks: `undersampled_data = under_sampling.TomekLinks().fit_resample(X, y)`
- Perform multi-class combination of over- and undersampling with SMOTEENN: `resampled_data = combine.SMOTEENN(sampling_strategy='auto').fit_resample(X, y)`

14. Handling Imbalanced Time Series Data

- Resample time series data using sliding window: `resampled_data = series_to_supervised(data, n_in=1, n_out=1)`
- Perform time-based splitting of data: `X_train, X_test, y_train, y_test = temporal_train_test_split(X, y, test_size=0.2)`
- Perform time-based cross-validation: `scores = temporal_cross_val_score(clf, X, y, cv=TimeSeriesSplit(n_splits=5))`
- Perform rolling window cross-validation: `scores = rolling_cross_val_score(clf, X, y, window_size=30, step_size=1)`
- Apply oversampling within each time window: `resampled_data = over_sampling.SMOTE().fit_resample(X_window, y_window)`

15. Advanced Techniques

- Perform anomaly detection using Isolation Forest: `clf = ensemble.IsolationForest(contamination=0.1)`
- Perform anomaly detection using Local Outlier Factor: `clf = neighbors.LocalOutlierFactor(n_neighbors=20, contamination=0.1)`
- Perform anomaly detection using One-Class SVM: `clf = svm.OneClassSVM(nu=0.1)`
- Perform synthetic data generation using Variational Autoencoder (VAE): `vae = keras.Sequential([keras.layers.Dense(32, input_shape=(num_features,), activation='relu'), keras.layers.Dense(16, activation='relu'), keras.layers.Dense(2, activation='linear'), keras.layers.Dense(16, activation='relu'), keras.layers.Dense(32, activation='relu'), keras.layers.Dense(num_features, activation='sigmoid')])`

- Perform synthetic data generation using Generative Adversarial Network (GAN): `generator = keras.Sequential([keras.layers.Dense(128, input_shape=(latent_dim,), activation='relu'), keras.layers.Dense(256, activation='relu'), keras.layers.Dense(num_features, activation='sigmoid')]); discriminator = keras.Sequential([keras.layers.Dense(256, input_shape=(num_features,), activation='relu'), keras.layers.Dense(128, activation='relu'), keras.layers.Dense(1, activation='sigmoid')])`
- Perform active learning with uncertainty sampling: `clf.fit(X_train, y_train); uncertain_samples = clf.predict_proba(X_pool).max(axis=1); query_idx = uncertain_samples.argsort()[:n_queries]`
- Perform active learning with query-by-committee: `committee = [svm.SVC(C=1, kernel='linear'), svm.SVC(C=1, kernel='rbf'), svm.SVC(C=10, kernel='linear')]; predictions = np.array([model.predict(X_pool) for model in committee]); disagreement = np.sum(predictions != predictions[0], axis=0); query_idx = disagreement.argsort()[::-1][:n_queries]`