[Regression Analysis] [cheatsheet]

Data Preparation

- **Load dataset:** `import pandas as pd; data = pd.read_csv('data.csv')`
- **Handle missing values:** `data.fillna(data.mean(), inplace=True)`
- **Feature selection (Correlation):** `correlation = data.corr()`
- **One-hot encoding:** `pd.get_dummies(data)`
- **Feature scaling (Standardization):** `from sklearn.preprocessing import StandardScaler; scaler = StandardScaler(); scaled_data = scaler.fit_transform(data)`
- **Feature scaling (Normalization):** `from sklearn.preprocessing import MinMaxScaler; scaler = MinMaxScaler(); normalized_data = scaler.fit_transform(data)`
- **Split dataset:** `from sklearn.model_selection import train_test_split; X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)`
- **Polynomial feature generation:** `from sklearn.preprocessing import PolynomialFeatures; poly = PolynomialFeatures(degree=3); X_poly = poly.fit_transform(X)`

Regression Model Selection

- **Linear Regression:** `from sklearn.linear_model import LinearRegression; model = LinearRegression()`
- **Ridge Regression:** `from sklearn.linear_model import Ridge; model = Ridge(alpha=1.0)`
- **Lasso Regression:** `from sklearn.linear_model import Lasso; model = Lasso(alpha=0.1)`
- **ElasticNet:** `from sklearn.linear_model import ElasticNet; model = ElasticNet(alpha=0.1, l1_ratio=0.5)`
- **Logistic Regression:** `from sklearn.linear_model import LogisticRegression; model = LogisticRegression()`
- **Polynomial Regression:** `# Use PolynomialFeatures in combination with LinearRegression`
- **Decision Tree Regression:** `from sklearn.tree import DecisionTreeRegressor; model = DecisionTreeRegressor()`

By: Waleed Mousa

- **Random Forest Regression:** `from sklearn.ensemble import RandomForestRegressor; model = RandomForestRegressor()`
- **Support Vector Regression:** `from sklearn.svm import SVR; model = SVR()`
- **K-Nearest Neighbors Regression:** `from sklearn.neighbors import KNeighborsRegressor; model = KNeighborsRegressor(n_neighbors=5)`

Model Fitting

- **Fit model:** `model.fit(X_train, y_train)`
- **Predict values:** `predictions = model.predict(X_test)`
- **Calculate R-squared:** `model.score(X_test, y_test)`
- **Coefficient of determination:** `from sklearn.metrics import r2_score; r2_score(y_test, predictions)`
- **Mean Squared Error (MSE):** `from sklearn.metrics import mean_squared_error; mse = mean_squared_error(y_test, predictions)`
- **Root Mean Squared Error (RMSE):** `import numpy as np; rmse = np.sqrt(mse)`
- **Mean Absolute Error (MAE):** `from sklearn.metrics import mean_absolute_error; mae = mean_absolute_error(y_test, predictions)`
- **Model coefficients:** `coefficients = model.coef_`
- **Model intercept:** `intercept = model.intercept_`
- **Cross-validation:** `from sklearn.model_selection import cross_val_score; scores = cross_val_score(model, X, y, cv=5)`

Diagnostics and Model Evaluation

- **Plot residuals:** `import matplotlib.pyplot as plt; residuals = y_test - predictions; plt.scatter(y_test, residuals)`
- **Check for homoscedasticity:** `plt.scatter(predictions, residuals)`
- **Q-Q plot for normality of residuals:** `import scipy.stats as stats; stats.probplot(residuals, dist="norm", plot=plt)`
- **Calculate AIC:** `from statsmodels.regression.linear_model import OLS; model = OLS(y, X); result = model.fit(); result.aic`
- **Calculate BIC:** `result.bic`
- **Feature importance (for tree-based models):** `importance = model.feature_importances_`

- **Confusion matrix (for logistic regression):** `from sklearn.metrics import confusion_matrix; cm = confusion_matrix(y_test, predictions)`
- **Classification report (for logistic regression):** `from sklearn.metrics import classification_report; report = classification_report(y_test, predictions)`
- **ROC Curve (for logistic regression):** `from sklearn.metrics import roc_curve; fpr, tpr, thresholds = roc_curve(y_test, model.predict_proba(X_test)[:,1])`
- **Precision-Recall Curve:** `from sklearn.metrics import precision_recall_curve; precision, recall, thresholds = precision_recall_curve(y_test, model.predict_proba(X_test)[:,1])`

Advanced Techniques and Considerations

- **Feature selection with RFE:** `from sklearn.feature_selection import RFE; selector = RFE(model, n_features_to_select=5); selector = selector.fit(X, y)`
- **Hyperparameter tuning with GridSearchCV:** `from sklearn.model_selection import GridSearchCV; parameters = {'alpha':[0.1, 1, 10]}; grid = GridSearchCV(model, parameters, cv=5); grid.fit(X, y)`
- **Regularization path (for Lasso/Ridge):** `from sklearn.linear_model import lasso_path; alphas, coefs, _ = lasso_path(X, y, alphas=[0.1, 1, 10])`
- **Learning curve:** `from sklearn.model_selection import learning_curve; train_sizes, train_scores, test_scores = learning_curve(model, X, y, cv=5)`
- **Validation curve:** `from sklearn.model_selection import validation_curve; param_range = np.logspace(-6, -1, 5); train_scores, test_scores = validation_curve(model, X, y, param_name="alpha", param_range=param_range, cv=5)`
- **Partial dependence plots (for ensemble models):** `from sklearn.inspection import plot_partial_dependence; plot_partial_dependence(model, X, [0, 1])`

Data Transformation and Interaction Effects

- **Log transformation of a feature:** `data['log_feature'] = np.log(data['feature'])`

- **Square root transformation:** `data['sqrt_feature'] = np.sqrt(data['feature'])`
- **Box-Cox transformation:** `from scipy.stats import boxcox; data['boxcox_feature'], _ = boxcox(data['feature'])`
- **Creating interaction terms manually:** `data['interaction'] = data['feature1'] * data['feature2']`
- **Automatic interaction terms with PolynomialFeatures:** `from sklearn.preprocessing import PolynomialFeatures; poly = PolynomialFeatures(interaction_only=True); data_interaction = poly.fit_transform(data)`

Ensemble Methods and Model Improvement

- **Gradient Boosting Regression:** `from sklearn.ensemble import GradientBoostingRegressor; model = GradientBoostingRegressor()`
- **XGBoost Regression:** `from xgboost import XGBRegressor; model = XGBRegressor()`
- **LightGBM Regression:** `from lightgbm import LGBMRegressor; model = LGBMRegressor()`
- **Stacking models:** `from sklearn.ensemble import StackingRegressor; estimators = [('lr', LinearRegression()), ('svr', SVR())]; model = StackingRegressor(estimators=estimators)`
- **Bagging with Random Forests:** `# Random Forests inherently use bagging`

Dealing with Non-linear Relationships

- **Kernel Ridge Regression:** `from sklearn.kernel_ridge import KernelRidge; model = KernelRidge(kernel='polynomial', degree=2)`
- **SVM with non-linear kernel:** `model = SVR(kernel='rbf')`
- **Non-linear transformation of target variable (log):** `y_log = np.log(y)`
- **GAMs for flexible non-linear modeling:** `from pygam import LinearGAM, s; gam = LinearGAM(s(0) + s(1)).fit(X, y)`

Model Comparison and Selection

- **Akaike Information Criterion (AIC) for model comparison:** `# Refer to operation 32 for calculation method`

- **Bayesian Information Criterion (BIC) for model comparison:** `# Refer to operation 33 for calculation method`
- **Adjusted R-squared for model comparison:** `1 - (1-model.score(X, y))*(len(y)-1)/(len(y)-X.shape[1]-1)`
- **F-test to compare models:** `from sklearn.feature_selection import f_regression; F, p_values = f_regression(X, y)`

Advanced Diagnostics

- **VIF (Variance Inflation Factor) for multicollinearity:** `from statsmodels.stats.outliers_influence import variance_inflation_factor; VIF = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]`
- **Durbin-Watson test for autocorrelation:** `from statsmodels.stats.stattools import durbin_watson; dw = durbin_watson(residuals)`
- **Cook's distance for influence points:** `from statsmodels.stats.outliers_influence import OLSInfluence; influence = OLSInfluence(model); cooks = influence.cooks_distance[0]`
- **Leverage to identify influential observations:** `leverage = influence.hat_matrix_diag`

Prediction and Validation

- **Predict with confidence intervals:** `# For linear models, use statsmodels for prediction: predictions, intervals = model.get_prediction(X_new).summary_frame(alpha=0.05)`
- **Bootstrap resampling for estimating prediction uncertainty:** `from sklearn.utils import resample; bootstrapped_samples = resample(predictions, n_samples=1000)`
- **Permutation importance for feature evaluation:** `from sklearn.inspection import permutation_importance; result = permutation_importance(model, X_test, y_test, n_repeats=10)`
- **Shapley values for feature impact:** `import shap; explainer = shap.TreeExplainer(model); shap_values = explainer.shap_values(X)`

Post-modeling Analysis

- **Model summary with statsmodels:** `import statsmodels.api as sm; model = sm.OLS(y, sm.add_constant(X)); results = model.fit(); print(results.summary())`
- **Partial dependence plots for feature effect visualization:** `# Refer to operation 44 for sklearn or use 'plot_partial_dependance' from the appropriate library for advanced models`
- **ICE plots for individual conditional expectations:** `from pycebox.ice import ice, ice_plot; ice_df = ice(data, 'feature', model.predict); ice_plot(ice_df)`
- **LIME for local interpretation:** `import lime; import lime.lime_tabular; explainer = lime.lime_tabular.LimeTabularExplainer(training_data=X_train, feature_names=X.columns, class_names=['target'], mode='regression'); explanation = explainer.explain_instance(data_row=X_test.iloc[0], predict_fn=model.predict)`
- **Model persistence with joblib:** `from joblib import dump, load; dump(model, 'model.joblib'); model = load('model.joblib')`

Handling Categorical Variables

- **Ordinal encoding:** `from sklearn.preprocessing import OrdinalEncoder; encoder = OrdinalEncoder(); data['encoded_feature'] = encoder.fit_transform(data[['feature']])`
- **Frequency encoding:** `frequency = data['feature'].value_counts() / len(data); data['freq_encoded_feature'] = data['feature'].map(frequency)`
- **Target encoding:** `import category_encoders as ce; encoder = ce.TargetEncoder(); data['target_encoded_feature'] = encoder.fit_transform(data['feature'], data['target'])`

Enhancing Model Performance

- **Feature engineering:** `data['new_feature'] = data['feature1'] / data['feature2']`
- **Removing outliers:** `from scipy import stats; data = data[(np.abs(stats.zscore(data['feature'])) < 3)]`
- **Smoothing noisy data (Moving Average):** `data['smoothed_feature'] = data['feature'].rolling(window=5).mean()`

- **Dimensionality reduction (PCA):** `from sklearn.decomposition import PCA; pca = PCA(n_components=2); X_pca = pca.fit_transform(X)`
- **Clustering as a feature (K-Means):** `from sklearn.cluster import KMeans; kmeans = KMeans(n_clusters=3); data['cluster'] = kmeans.fit_predict(data[['feature1', 'feature2']])`
- **Using external data for additional features:** `# Assume external_data is loaded; data = pd.merge(data, external_data, on='key')`

Advanced Diagnostics and Model Analysis

- **Cross-validation with multiple metrics:** `from sklearn.model_selection import cross_validate; scoring = ['r2', 'neg_mean_squared_error']; results = cross_validate(model, X, y, scoring=scoring)`
- **Time series cross-validation:** `from sklearn.model_selection import TimeSeriesSplit; tscv = TimeSeriesSplit(); for train_index, test_index in tscv.split(X): ...`
- **Spatial cross-validation (for geographical data):** `from sklearn.model_selection import GroupShuffleSplit; gss = GroupShuffleSplit(test_size=.3, n_splits=1, random_state=42).split(X, groups=X['group'])`
- **Analyzing residuals for patterns:** `plt.plot(y_test, residuals, marker='o', linestyle='')`
- **Testing for stationarity in residuals (ADF test):** `from statsmodels.tsa.stattools import adfuller; adf_result = adfuller(residuals)`
- **Model stability testing (bootstrap):** `# Refer to operation 68 for bootstrap resampling`

Advanced Prediction Techniques

- **Forecasting with ARIMA (for time series):** `from statsmodels.tsa.arima.model import ARIMA; model = ARIMA(data['feature'], order=(1,1,1)); result = model.fit()`
- **Using Prophet for time series prediction:** `from fbprophet import Prophet; m = Prophet(); m.fit(data); future = m.make_future_dataframe(periods=365); forecast = m.predict(future)`

- **Multi-output regression:** `from sklearn.multioutput import MultiOutputRegressor; mor = MultiOutputRegressor(model).fit(X_train, y_train_multi)`
- **Quantile regression for prediction intervals:** `import statsmodels.formula.api as smf; model = smf.quantreg('y ~ X', data).fit(q=0.5)`

Model Interpretation and Explanation

- **Advanced SHAP value interpretation:** `shap.summary_plot(shap_values, X, plot_type="bar")`
- **ALE (Accumulated Local Effects) plots for feature effects:** `from alibi.explainers import ALE, plot_ale; ale = ALE(model.predict, feature_names=X.columns); ale_exp = ale.explain(X.values); plot_ale(ale_exp)`
- **Global model explanation with Skater:** `from skater.core.explanations import Interpretation; from skater.model import InMemoryModel; interpreter = Interpretation(X_test, feature_names=X.columns); model = InMemoryModel(model.predict, examples=X_train); plots = interpreter.feature_importance.plot_feature_importance(model, ascending=False)`
- **Decision tree visualization for simple models:** `from sklearn.tree import plot_tree; plot_tree(decision_tree_model); plt.show()`
- **Visualizing feature interactions with PDPBox:** `from pdpbox import pdp; pdp_interact = pdp.pdp_interact(model, dataset=X, model_features=X.columns, features=['feature1', 'feature2']); pdp.pdp_interact_plot(pdp_interact, ['feature1', 'feature2'], plot_type='contour')`
- **Visualizing SVM decision boundaries:** `from mlxtend.plotting import plot_decision_regions; plot_decision_regions(X.values, y.values, clf=svm_model, legend=2)`
- **Visualizing K-Means clustering boundaries:** `# Assume data is 2D for visualization; plt.scatter(data[:,0], data[:,1], c=kmeans.labels_); centers = kmeans.cluster_centers_; plt.scatter(centers[:,0], centers[:,1], c='red', s=200, alpha=0.5);`
- **Visualizing embeddings with t-SNE:** `from sklearn.manifold import TSNE; tsne = TSNE(n_components=2); X_tsne = tsne.fit_transform(X)`

- **Exploring model errors:** error_indices = np.where(y_test != predictions)[0]; wrong_predictions = X_test.iloc[error_indices]
- **Visualizing regression diagnostics with Yellowbrick:** from yellowbrick.regressor import ResidualsPlot; visualizer = ResidualsPlot(model); visualizer.fit(X_train, y_train); visualizer.score(X_test, y_test); visualizer.show()
- **Model comparison with scikit-plot:** import scikitplot as skplt; skplt.estimators.plot_learning_curve(model1, X, y); skplt.estimators.plot_learning_curve(model2, X, y)