[Model Debugging & Testing] (CheatSheet)

Data Validation

- **Check for missing values**: `data.isnull().sum()`
- **Validate data types**: `data.dtypes`
- **Check for class imbalance**: `data['label'].value_counts()`
- **Validate range of values**: `data.describe()`
- **Detect outliers (IQR)**: `Q1 = data.quantile(0.25); Q3 = data.quantile(0.75); IQR = Q3 - Q1; outliers = data[(data < (Q1 - 1.5 * IQR)) | (data > (Q3 + 1.5 * IQR))]`
- **Check for data duplication**: `data.duplicated().sum()`
- **Validate against schema (Pandas)**: `schema = pd.io.json.build_table_schema(data); pd.io.json.validate(data, schema)`
- **Check for consistent labeling**: `data['label'].unique()`
- **Visualize data distribution (Seaborn)**: `sns.distplot(data['feature'])`
- **Correlation matrix**: `data.corr()`
- **Feature importance visualization**: `pd.Series(model.feature_importances_, index=features).nlargest(10).plot(kind='barh')`
- **Cross-tabulation of categories**: `pd.crosstab(data['feature1'], data['feature2'])`
- **Plot missing data heatmap**: `sns.heatmap(data.isnull(), cbar=False)`
- **Time series data check for seasonality**: `pd.plotting.autocorrelation_plot(data['time_series_feature'])`
- **Ensure data is shuffled for training**: `data = data.sample(frac=1).reset_index(drop=True)`

Model Validation

- **Cross-validation**: `cross_val_score(model, X, y, cv=5)`
- **Train/test split**: `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)`
- **Learning curves (Scikit-learn)**: `plot_learning_curve(model, X, y, cv=5)`
- **Validation curves (Scikit-learn)**: `plot_validation_curve(model, X, y, param_name, param_range, cv=5)`
- **Hyperparameter tuning (GridSearchCV)**: `GridSearchCV(model, param_grid, cv=5).fit(X, y)`
- **Check model assumptions**: `sm.OLS(y, sm.add_constant(X)).fit().summary()`
- **Performance on unseen data**: `model.predict(X_new)`

By: Waleed Mousa

- **Bootstrap sampling for model stability**: `bootstrap = resample(data, replace=True, n_samples=100, random_state=1)`
- **K-Fold cross-validation with stratification**: `StratifiedKFold(n_splits=5).split(X, y)`
- **Use A/B testing for model performance comparison**: `ttest_ind(groupA['metric'], groupB['metric'])`
- **Model comparison with statistical significance**: `stats.f_oneway(model1_scores, model2_scores)`
- **ROC-AUC curve for binary classification**: `plot_roc_curve(model, X_test, y_test)`
- **Precision-Recall curve for imbalanced datasets**: `plot_precision_recall_curve(model, X_test, y_test)`
- **Confusion matrix visualization**: `plot_confusion_matrix(model, X_test, y_test)`
- **Feature permutation importance**: `permutation_importance(model, X_val, y_val, n_repeats=30)`

Debugging Techniques

- **Log model predictions and actuals**: `log.info(f"Predictions: {predictions}, Actuals: {y_test}")`
- **Use assert statements to check shapes**: `assert X_train.shape[0] == y_train.shape[0]`
- **Check for NaNs in predictions**: `np.isnan(predictions).any()`
- **Visualize model decisions**: `plot_decision_regions(X, y, clf=model)`
- **Check weight and bias values**: `print(model.coef_, model.intercept_)`
- **Track gradients in neural networks (TensorFlow)**: `tf.debugging.check_numerics(tensor, message='Check failed')`
- **Profile model training (TensorFlow)**: `tf.profiler.experimental.start('logdir'); tf.profiler.experimental.stop()`
- **Monitor layer outputs (Keras Callbacks)**: `ModelCheckpoint(filepath='model.h5', monitor='val_loss')`
- **Use Explainable AI frameworks for insights**: `shap_values = shap.TreeExplainer(model).shap_values(X_train)`
- **Detect and log model drift**: `if drift_detected: logger.warning('Model drift detected')`
- **Unit tests for data pipelines**: `def test_pipeline(): assert processed_data.shape[0] > 0`
- **Logging hyperparameters and results**: `mlflow.log_params(params); mlflow.log_metric("accuracy", accuracy)`

- **Memory profiling for large models**: `from memory_profiler import profile; @profile def train_model(): model.fit(X, y)`
- **CPU/GPU utilization monitoring**: `nvidia-smi` (for NVIDIA GPUs)
- **Validate model input features range**: `assert X.min() >= feature_range[0] and X.max() <= feature_range[1]`

Performance Testing

- **Load testing models**: `timeit.timeit(lambda: model.predict(X_test), number=1000)`
- **Stress testing models under heavy loads**: `stress_test_model(model, X, y, n_iterations=10000)`
- **Benchmarking model inference time**: `start_time = time.time(); model.predict(X_test); print(time.time() - start_time)`
- **Compare model performance across hardware**: `benchmark_on_cpu_gpu(model, X_test)`
- **Testing model resilience to adversarial attacks**: `test_adversarial_resilience(model, X_test, y_test)`
- **Scalability testing of model pipelines**: `scalability_test(model_pipeline, data_size_range)`
- **Memory usage of model during inference**: `memory_usage = memory_profiler.memory_usage((model.predict, (X_test,)))`
- **Test model with synthetic data for edge cases**: `test_with_synthetic_data(model)`
- **Concurrency testing with multiple requests**: `concurrent_inference_test(model, X_test, n_concurrent_requests=50)`
- **Latency testing at different data scales**: `latency_test(model, X_test_sizes)`

Interpretability and Explainability

- **Feature importance (Scikit-learn)**: `model.feature_importances_`
- **Partial dependence plots (PDPbox)**: `pdp.pdp_plot(pdp.pdp_isolate(model, dataset, model_features, feature), feature)`
- **Local Interpretable Model-agnostic Explanations (LIME)**: `lime_explainer.explain_instance(data_row, model.predict_proba).as_pyplot_figure()`
- **SHAP values**: `shap_values = shap.TreeExplainer(model).shap_values(X_test)`
- **Global Surrogate Models**: `surrogate = DecisionTreeClassifier().fit(X, model.predict(X))`

- **Anchor explanations for robust predictions**: `anchor_explainer.explain_instance(data_row, model.predict, threshold=0.95)`
- **Feature interaction detection (Scikit-learn)**: `interaction = interaction_terms(model, X, y)`
- **Visualize model decision boundaries**: `plot_decision_boundaries(X, y, model)`
- **Model agnostic metric plots**: `metric_plot.compare_models(true_y, model1_y, model2_y)`
- **Visualize embeddings and clusters (t-SNE, PCA)**: `tsne_plot(model_embeddings)`
- **Counterfactual explanations**: `counterfactual = find_counterfactual(instance, model.predict, desired_label=1)`
- **Use AI Explainability 360 toolkit for comprehensive insights**: `aix360.explain(model, data)`
- **Diagnose model with What-If Tool**: `WitConfigBuilder(test_examples).set_model_type('classification').set_target_feature('label')`
- **Model Cards for model transparency**: `generate_model_card(model_details, performance_metrics, ethical_considerations)`
- **Fairness assessment in model predictions**: `fairness_indicators = compute_fairness_metrics(y_true, y_pred, sensitive_features)`

Error Analysis

- **Analyze error types**: `error_types = classify_errors(y_test, predictions)`
- **Plot error distribution**: `plt.hist(predictions - y_test)`
- **Review misclassified examples**: `misclassified = X_test[predictions != y_test]`
- **Analyze errors by category**: `error_summary_by_category(y_test, predictions)`
- **Use confusion matrix for multi-class error analysis**: `sns.heatmap(confusion_matrix(y_test, predictions), annot=True)`
- **Text data error token analysis**: `token_error_analysis(misclassified_texts)`
- **Regression model residual analysis**: `sns.residplot(x=predicted_values, y=actual_values - predicted_values)`
- **Analyze model errors over time**: `plot_error_trends(errors_over_time)`
- **Feature contribution to errors analysis (SHAP values)**: `shap.summary_plot(shap_values[misclassified], features[misclassified])`
- **Error bucketing for targeted analysis**: `bucket_errors(errors, criteria='magnitude')`

Adversarial Testing

- **Generate adversarial examples (Text)**: `adversarial_text = perturb_text(original_text)`
- **Test model against adversarial examples**: `model_performance_on_adversarial = model.evaluate(adversarial_examples, y_true)`
- **Image data adversarial example generation**: `adversarial_image = create_adversarial_example(model, original_image)`
- **Use adversarial robustness toolbox (ART) for testing**: `art_attacks = art.attacks.FastGradientMethod(classifier)`
- **Evaluate model robustness to adversarial attacks**: `robustness = calculate_model_robustness(model, X_test, y_test)`
- **Adversarial retraining for model improvement**: `model_retrained = retrain_model_with_adversarials(model, adversarial_examples)`
- **Visual inspection of adversarial examples**: `plot_comparison(original, adversarial)`
- **Benchmarking model against common adversarial attacks**: `benchmark_adversarial_defenses(model)`
- **Adversarial example detection mechanism**: `is_adversarial = detect_adversarial_activity(input_data)`
- **Implement model defenses against adversarial attacks**: `defended_model = apply_defenses(original_model)`

Model Monitoring and Updating

- **Set up model performance dashboards**: `create_dashboard(model_metrics_over_time)`
- **Automate regular model evaluation**: `schedule_model_evaluation(model, data_stream, frequency='weekly')`
- **Monitor data drift and concept drift**: `detect_drift(data_stream, model)`
- **Alerting for performance degradation**: `setup_alerts_for_degradation(model_performance_metrics)`
- **Version control for models and datasets**: `dvc.track_changes(model_file, data_file)`
- **Rollback to previous model versions if needed**: `model = load_model_version(version_number)`
- **Automatically retrain model on new data**: `model_updated = auto_retrain(model, new_data)`
- **Continuous integration and deployment for models (CI/CD)**: `setup_ci_cd_pipeline_for_model_deployment()`

- **A/B testing for new model versions**: `perform_ab_testing(model_A, model_B, test_data)`
- **Track and log all model experiments**: `log_experiment_details(experiment_id, experiment_params, results)`
- **Use model explainability as a monitoring tool**: `monitor_model_with_shap(model, data_stream)`
- **Model performance benchmarking across different environments**: `benchmark_model_across_environments(model)`
- **Automate model health checks**: `setup_automatic_health_checks(model)`
- **Deploy shadow models for live performance comparison**: `deploy_shadow_model(original_model, candidate_model)`
- **Feedback loop for model learning from new data**: `implement_feedback_loop(model, operational_data)`