[300+ One-Liner Operations For ML Engineers] CheatSheet

ML Engineer Stack CheatSheet

🐼 PANDAS DATA MANIPULATION

Loading Data

- Load CSV: `df = pd.read_csv("data.csv")`
- Load CSV with specific columns: `df = pd.read_csv("data.csv", usecols=["col1", "col2"])`
- Load CSV with index: `df = pd.read_csv("data.csv", index_col="id")`
- Load CSV skip rows: `df = pd.read_csv("data.csv", skiprows=5)`
- Load CSV with dtypes: `df = pd.read_csv("data.csv", dtype={"col": "int32"})`
- Load Excel: `df = pd.read_excel("data.xlsx", sheet_name="Sheet1")`
- Load JSON: `df = pd.read_json("data.json")`
- Load from SQL: `df = pd.read_sql("SELECT * FROM table", connection)`
- Load from dictionary: `df = pd.DataFrame({"col1": [1, 2], "col2": [3, 4]})`
- Load from clipboard: `df = pd.read_clipboard()`

DataFrame Inspection

- View first rows: `df.head(10)`
- View last rows: `df.tail(10)`
- View random sample: `df.sample(5)`
- View shape: `df.shape`
- View columns: `df.columns.tolist()`
- View dtypes: `df.dtypes`
- View info: `df.info()`
- View statistics: `df.describe()`
- View memory usage: `df.memory_usage(deep=True)`
- View unique counts: `df.nunique()`

Selecting Data

- Select column: `df["column"]`
- Select multiple columns: `df[["col1", "col2"]]`
- Select by index: `df.iloc[0:5]`
- Select by label: `df.loc["row_label"]`
- Select by condition: `df[df["col"] > 0]`
- Select multiple conditions: `df[(df["col1"] > 0) & (df["col2"] < 10)]`
- Select with isin: `df[df["col"].isin(["a", "b", "c"])]`
- Select with query: `df.query("col1 > 0 and col2 < 10")`
- Select columns by dtype: `df.select_dtypes(include=["number"])`
- Select columns by pattern: `df.filter(regex="^feature_")`

Data Cleaning

- Drop duplicates: `df.drop_duplicates()`
- Drop duplicates by column: `df.drop_duplicates(subset=["col1"])`
- Drop missing values: `df.dropna()`
- Drop columns with missing: `df.dropna(axis=1)`
- Fill missing with value: `df.fillna(0)`

By: Waleed Mousa

- Fill missing with mean: `df["col"].fillna(df["col"].mean())`
- Fill missing forward: `df.fillna(method="ffill")`
- Fill missing backward: `df.fillna(method="bfill")`
- Replace values: `df.replace({"old": "new"})`
- Clip outliers: `df["col"].clip(lower=0, upper=100)`

Data Transformation

- Rename columns: `df.rename(columns={"old": "new"})`
- Rename columns lowercase: `df.columns = df.columns.str.lower()`
- Add new column: `df["new_col"] = df["col1"] + df["col2"]`
- Apply function: `df["col"].apply(lambda x: x * 2)`
- Apply to dataframe: `df.apply(lambda row: row["a"] + row["b"], axis=1)`
- Map values: `df["col"].map({"a": 1, "b": 2})`
- Bin continuous: `pd.cut(df["col"], bins=5)`
- Quantile bins: `pd.qcut(df["col"], q=4, labels=["Q1", "Q2", "Q3", "Q4"])`
- One-hot encode: `pd.get_dummies(df, columns=["cat_col"])`
- Label encode: `df["col"].astype("category").cat.codes`

Aggregation & Grouping

- Group by single column: `df.groupby("col").mean()`
- Group by multiple columns: `df.groupby(["col1", "col2"]).sum()`
- Group with multiple aggs: `df.groupby("col").agg({"val": ["mean", "sum", "count"]})`
- Group with named aggs: `df.groupby("col").agg(avg=("val", "mean"), total=("val", "sum"))`
- Transform within group: `df.groupby("col")["val"].transform("mean")`
- Rank within group: `df.groupby("col")["val"].rank()`
- Cumulative sum by group: `df.groupby("col")["val"].cumsum()`
- Pivot table: `df.pivot_table(values="val", index="row", columns="col", aggfunc="mean")`
- Crosstab: `pd.crosstab(df["col1"], df["col2"])`
- Value counts: `df["col"].value_counts(normalize=True)`

Merging & Joining

- Inner join: `pd.merge(df1, df2, on="key")`
- Left join: `pd.merge(df1, df2, on="key", how="left")`
- Right join: `pd.merge(df1, df2, on="key", how="right")`
- Outer join: `pd.merge(df1, df2, on="key", how="outer")`
- Join on multiple keys: `pd.merge(df1, df2, on=["key1", "key2"])`
- Join on different names: `pd.merge(df1, df2, left_on="a", right_on="b")`
- Concatenate rows: `pd.concat([df1, df2], axis=0)`
- Concatenate columns: `pd.concat([df1, df2], axis=1)`
- Append rows: `df1._append(df2, ignore_index=True)`
- Merge with indicator: `pd.merge(df1, df2, on="key", indicator=True)`

🔢 NUMPY OPERATIONS

Array Creation

- Create array: `np.array([1, 2, 3, 4, 5])`
- Create zeros: `np.zeros((3, 4))`
- Create ones: `np.ones((3, 4))`
- Create range: `np.arange(0, 10, 2)`
- Create linspace: `np.linspace(0, 1, 100)`
- Create identity: `np.eye(3)`
- Create diagonal: `np.diag([1, 2, 3])`
- Create random: `np.random.rand(3, 4)`
- Create random normal: `np.random.randn(3, 4)`
- Create random integers: `np.random.randint(0, 10, (3, 4))`

Array Manipulation

- Reshape array: `arr.reshape(3, 4)`
- Flatten array: `arr.flatten()`
- Transpose: `arr.T`
- Concatenate: `np.concatenate([arr1, arr2], axis=0)`
- Stack vertically: `np.vstack([arr1, arr2])`
- Stack horizontally: `np.hstack([arr1, arr2])`
- Split array: `np.split(arr, 3)`
- Add dimension: `arr[np.newaxis, :]`
- Squeeze dimension: `np.squeeze(arr)`
- Tile array: `np.tile(arr, (2, 3))`

Array Math

- Element-wise add: `arr1 + arr2`
- Element-wise multiply: `arr1 * arr2`
- Matrix multiply: `np.dot(arr1, arr2)`
- Matrix multiply @: `arr1 @ arr2`
- Sum all: `np.sum(arr)`
- Sum by axis: `np.sum(arr, axis=0)`
- Mean: `np.mean(arr)`
- Standard deviation: `np.std(arr)`
- Min/Max: `np.min(arr)`, `np.max(arr)`
- Argmin/Argmax: `np.argmin(arr)`, `np.argmax(arr)`

Array Indexing

- Boolean indexing: `arr[arr > 0]`
- Fancy indexing: `arr[[0, 2, 4]]`
- Where condition: `np.where(arr > 0, arr, 0)`
- Nonzero indices: `np.nonzero(arr)`
- Clip values: `np.clip(arr, 0, 1)`
- Unique values: `np.unique(arr)`
- Sort array: `np.sort(arr)`
- Argsort: `np.argsort(arr)`
- Percentile: `np.percentile(arr, 95)`
- Histogram: `np.histogram(arr, bins=10)`

By: Waleed Mousa

🧠 SCIKIT-LEARN

Data Preprocessing

- Train test split: `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)`
- Stratified split: `train_test_split(X, y, test_size=0.2, stratify=y)`
- Standard scaler: `scaler = StandardScaler(); X_scaled = scaler.fit_transform(X)`
- Min-max scaler: `scaler = MinMaxScaler(); X_scaled = scaler.fit_transform(X)`
- Robust scaler: `scaler = RobustScaler(); X_scaled = scaler.fit_transform(X)`
- Label encoder: `le = LabelEncoder(); y_encoded = le.fit_transform(y)`
- One-hot encoder: `ohe = OneHotEncoder(sparse=False); X_encoded = ohe.fit_transform(X)`
- Ordinal encoder: `oe = OrdinalEncoder(); X_encoded = oe.fit_transform(X)`
- Impute missing mean: `imputer = SimpleImputer(strategy="mean"); X_imputed = imputer.fit_transform(X)`
- Impute missing KNN: `imputer = KNNImputer(n_neighbors=5); X_imputed = imputer.fit_transform(X)`

Feature Engineering

- Polynomial features: `poly = PolynomialFeatures(degree=2); X_poly = poly.fit_transform(X)`
- Select K best: `selector = SelectKBest(f_classif, k=10); X_selected = selector.fit_transform(X, y)`
- Select percentile: `selector = SelectPercentile(f_classif, percentile=20); X_selected = selector.fit_transform(X, y)`
- RFE selection: `rfe = RFE(estimator, n_features_to_select=10); X_selected = rfe.fit_transform(X, y)`
- Variance threshold: `selector = VarianceThreshold(threshold=0.1); X_selected = selector.fit_transform(X)`
- PCA: `pca = PCA(n_components=0.95); X_pca = pca.fit_transform(X)`
- t-SNE: `tsne = TSNE(n_components=2); X_tsne = tsne.fit_transform(X)`
- Feature importance: `importances = model.feature_importances_`
- Permutation importance: `result = permutation_importance(model, X, y, n_repeats=10)`
- Correlation matrix: `corr_matrix = df.corr()`

Classification Models

- Logistic regression: `model = LogisticRegression(); model.fit(X_train, y_train)`
- Random forest: `model = RandomForestClassifier(n_estimators=100); model.fit(X_train, y_train)`
- Gradient boosting: `model = GradientBoostingClassifier(); model.fit(X_train, y_train)`
- SVM: `model = SVC(kernel="rbf"); model.fit(X_train, y_train)`

By: Waleed Mousa

- KNN: `model = KNeighborsClassifier(n_neighbors=5); model.fit(X_train, y_train)`
- Naive Bayes: `model = GaussianNB(); model.fit(X_train, y_train)`
- Decision tree: `model = DecisionTreeClassifier(); model.fit(X_train, y_train)`
- XGBoost: `model = XGBClassifier(); model.fit(X_train, y_train)`
- LightGBM: `model = LGBMClassifier(); model.fit(X_train, y_train)`
- CatBoost: `model = CatBoostClassifier(verbose=0); model.fit(X_train, y_train)`

Regression Models

- Linear regression: `model = LinearRegression(); model.fit(X_train, y_train)`
- Ridge regression: `model = Ridge(alpha=1.0); model.fit(X_train, y_train)`
- Lasso regression: `model = Lasso(alpha=1.0); model.fit(X_train, y_train)`
- ElasticNet: `model = ElasticNet(alpha=1.0, l1_ratio=0.5); model.fit(X_train, y_train)`
- Random forest regressor: `model = RandomForestRegressor(); model.fit(X_train, y_train)`
- Gradient boosting regressor: `model = GradientBoostingRegressor(); model.fit(X_train, y_train)`
- SVR: `model = SVR(kernel="rbf"); model.fit(X_train, y_train)`
- XGBoost regressor: `model = XGBRegressor(); model.fit(X_train, y_train)`
- LightGBM regressor: `model = LGBMRegressor(); model.fit(X_train, y_train)`
- CatBoost regressor: `model = CatBoostRegressor(verbose=0); model.fit(X_train, y_train)`

Model Evaluation

- Accuracy: `accuracy_score(y_true, y_pred)`
- Precision: `precision_score(y_true, y_pred, average="weighted")`
- Recall: `recall_score(y_true, y_pred, average="weighted")`
- F1 score: `f1_score(y_true, y_pred, average="weighted")`
- ROC AUC: `roc_auc_score(y_true, y_proba)`
- Confusion matrix: `confusion_matrix(y_true, y_pred)`
- Classification report: `print(classification_report(y_true, y_pred))`
- MSE: `mean_squared_error(y_true, y_pred)`
- RMSE: `mean_squared_error(y_true, y_pred, squared=False)`
- MAE: `mean_absolute_error(y_true, y_pred)`
- R2 score: `r2_score(y_true, y_pred)`
- Cross validation: `scores = cross_val_score(model, X, y, cv=5)`
- Cross val predict: `y_pred = cross_val_predict(model, X, y, cv=5)`

Hyperparameter Tuning

- Grid search: `grid = GridSearchCV(model, param_grid, cv=5); grid.fit(X, y)`
- Random search: `search = RandomizedSearchCV(model, param_dist, n_iter=100, cv=5); search.fit(X, y)`
- Best params: `grid.best_params_`
- Best score: `grid.best_score_`

By: Waleed Mousa

- Best estimator: grid.best_estimator_
- Halving grid search: search = HalvingGridSearchCV(model, param_grid, cv=5); search.fit(X, y)
- Optuna integration: study = optuna.create_study(); study.optimize(objective, n_trials=100)
- Learning curve: train_sizes, train_scores, val_scores = learning_curve(model, X, y, cv=5)
- Validation curve: train_scores, val_scores = validation_curve(model, X, y, param_name, param_range, cv=5)

Pipelines

- Simple pipeline: pipe = Pipeline([("scaler", StandardScaler()), ("model", LogisticRegression())])
- Pipeline with column transformer: preprocessor = ColumnTransformer([("num", StandardScaler(), num_cols), ("cat", OneHotEncoder(), cat_cols)])
- Full pipeline: pipe = Pipeline([("preprocessor", preprocessor), ("model", RandomForestClassifier())])
- Pipeline fit: pipe.fit(X_train, y_train)
- Pipeline predict: y_pred = pipe.predict(X_test)
- Pipeline with feature selection: pipe = Pipeline([("scaler", StandardScaler()), ("selector", SelectKBest(k=10)), ("model", SVC())])
- Make pipeline shortcut: pipe = make_pipeline(StandardScaler(), PCA(n_components=10), LogisticRegression())
- Pipeline grid search: grid = GridSearchCV(pipe, {"model__C": [0.1, 1, 10]}, cv=5)
- Get pipeline step: pipe.named_steps["model"]
- Set pipeline params: pipe.set_params(model__C=10)

🔥 PYTORCH

Tensor Basics

- Create tensor: x = torch.tensor([1, 2, 3])
- Create zeros: x = torch.zeros(3, 4)
- Create ones: x = torch.ones(3, 4)
- Create random: x = torch.rand(3, 4)
- Create random normal: x = torch.randn(3, 4)
- Create range: x = torch.arange(0, 10, 2)
- Create linspace: x = torch.linspace(0, 1, 100)
- From numpy: x = torch.from_numpy(arr)
- To numpy: arr = x.numpy()
- To device: x = x.to("cuda")

Tensor Operations

- Reshape: x.view(2, 6)
- Reshape with -1: x.view(-1, 3)
- Transpose: x.T

By: Waleed Mousa

- Permute dims: `x.permute(2, 0, 1)`
- Squeeze: `x.squeeze()`
- Unsqueeze: `x.unsqueeze(0)`
- Concatenate: `torch.cat([x1, x2], dim=0)`
- Stack: `torch.stack([x1, x2], dim=0)`
- Split: `torch.split(x, 2, dim=0)`
- Clone: `x.clone()`

Tensor Math

- Matrix multiply: `torch.mm(x1, x2)`
- Batch matrix multiply: `torch.bmm(x1, x2)`
- Element-wise multiply: `x1 * x2`
- Sum: `x.sum()`
- Sum by dim: `x.sum(dim=0)`
- Mean: `x.mean()`
- Std: `x.std()`
- Max: `x.max()`
- Argmax: `x.argmax(dim=1)`
- Softmax: `torch.softmax(x, dim=1)`

Neural Network Layers

- Linear layer: `nn.Linear(in_features, out_features)`
- Conv2d: `nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)`
- MaxPool2d: `nn.MaxPool2d(kernel_size=2)`
- BatchNorm2d: `nn.BatchNorm2d(num_features)`
- Dropout: `nn.Dropout(p=0.5)`
- ReLU: `nn.ReLU()`
- LeakyReLU: `nn.LeakyReLU(0.2)`
- Sigmoid: `nn.Sigmoid()`
- Tanh: `nn.Tanh()`
- Softmax: `nn.Softmax(dim=1)`
- LSTM: `nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)`
- GRU: `nn.GRU(input_size, hidden_size, num_layers, batch_first=True)`
- Embedding: `nn.Embedding(num_embeddings, embedding_dim)`
- LayerNorm: `nn.LayerNorm(normalized_shape)`
- MultiheadAttention: `nn.MultiheadAttention(embed_dim, num_heads)`

Model Definition

- Sequential model: `model = nn.Sequential(nn.Linear(784, 256), nn.ReLU(), nn.Linear(256, 10))`
- Custom model class: `class Net(nn.Module): def __init__(self): super().__init__()`
- Forward pass: `def forward(self, x): return self.fc(x)`
- Get parameters: `model.parameters()`
- Count parameters: `sum(p.numel() for p in model.parameters())`
- Freeze parameters: `for param in model.parameters(): param.requires_grad = False`

By: Waleed Mousa

- Unfreeze parameters: `for param in model.parameters(): param.requires_grad = True`
- Move to GPU: `model.to("cuda")`
- Train mode: `model.train()`
- Eval mode: `model.eval()`

Loss Functions

- Cross entropy: `criterion = nn.CrossEntropyLoss()`
- Binary cross entropy: `criterion = nn.BCELoss()`
- BCE with logits: `criterion = nn.BCEWithLogitsLoss()`
- MSE loss: `criterion = nn.MSELoss()`
- L1 loss: `criterion = nn.L1Loss()`
- Smooth L1: `criterion = nn.SmoothL1Loss()`
- NLL loss: `criterion = nn.NLLLoss()`
- KL divergence: `criterion = nn.KLDivLoss()`
- Cosine embedding: `criterion = nn.CosineEmbeddingLoss()`
- Triplet margin: `criterion = nn.TripletMarginLoss()`

Optimizers

- SGD: `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)`
- Adam: `optimizer = optim.Adam(model.parameters(), lr=0.001)`
- AdamW: `optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)`
- RMSprop: `optimizer = optim.RMSprop(model.parameters(), lr=0.01)`
- Adagrad: `optimizer = optim.Adagrad(model.parameters(), lr=0.01)`
- Zero gradients: `optimizer.zero_grad()`
- Backward pass: `loss.backward()`
- Update weights: `optimizer.step()`
- Learning rate scheduler: `scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)`
- Cosine annealing: `scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)`

Training Loop

- Training step: `optimizer.zero_grad(); output = model(x); loss = criterion(output, y); loss.backward(); optimizer.step()`
- Validation step: `with torch.no_grad(): output = model(x); loss = criterion(output, y)`
- Get predictions: `_, preds = torch.max(output, 1)`
- Calculate accuracy: `accuracy = (preds == y).float().mean()`
- Gradient clipping: `torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)`
- Save model: `torch.save(model.state_dict(), "model.pth")`
- Load model: `model.load_state_dict(torch.load("model.pth"))`
- Save checkpoint: `torch.save({"epoch": epoch, "model": model.state_dict(), "optimizer": optimizer.state_dict()}, "checkpoint.pth")`

By: Waleed Mousa

- Load checkpoint: `checkpoint = torch.load("checkpoint.pth"); model.load_state_dict(checkpoint["model"])`
- Mixed precision: `with torch.cuda.amp.autocast(): output = model(x)`

Data Loading

- Custom dataset: `class MyDataset(Dataset): def __init__(self): ...; def __len__(self): ...; def __getitem__(self, idx): ...`
- DataLoader: `loader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)`
- Iterate loader: `for batch_idx, (data, target) in enumerate(loader): ...`
- TensorDataset: `dataset = TensorDataset(X_tensor, y_tensor)`
- Random split: `train_set, val_set = random_split(dataset, [0.8, 0.2])`
- Subset: `subset = Subset(dataset, indices)`
- Sampler: `sampler = WeightedRandomSampler(weights, num_samples)`
- Collate function: `loader = DataLoader(dataset, collate_fn=custom_collate)`
- Pin memory: `loader = DataLoader(dataset, pin_memory=True)`
- Prefetch factor: `loader = DataLoader(dataset, prefetch_factor=2)`

🤗 HUGGING FACE TRANSFORMERS

Loading Models

- Load model: `model = AutoModel.from_pretrained("bert-base-uncased")`
- Load tokenizer: `tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")`
- Load for classification: `model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)`
- Load for QA: `model = AutoModelForQuestionAnswering.from_pretrained("bert-base-uncased")`
- Load for NER: `model = AutoModelForTokenClassification.from_pretrained("bert-base-uncased")`
- Load for generation: `model = AutoModelForCausalLM.from_pretrained("gpt2")`
- Load for seq2seq: `model = AutoModelForSeq2SeqLM.from_pretrained("t5-base")`
- Load config: `config = AutoConfig.from_pretrained("bert-base-uncased")`
- Load from local: `model = AutoModel.from_pretrained("./my_model")`
- Load with custom config: `model = AutoModel.from_pretrained("bert-base-uncased", config=config)`

Tokenization

- Tokenize text: `tokens = tokenizer(text, return_tensors="pt")`
- Tokenize with padding: `tokens = tokenizer(texts, padding=True, return_tensors="pt")`
- Tokenize with truncation: `tokens = tokenizer(text, truncation=True, max_length=512)`
- Tokenize batch: `tokens = tokenizer(texts, padding=True, truncation=True, return_tensors="pt")`

By: Waleed Mousa

- Decode tokens: `text = tokenizer.decode(token_ids)`
- Batch decode: `texts = tokenizer.batch_decode(token_ids, skip_special_tokens=True)`
- Get vocab size: `vocab_size = tokenizer.vocab_size`
- Add special tokens: `tokenizer.add_special_tokens({"additional_special_tokens": ["[CUSTOM]"]})`
- Get token ID: `token_id = tokenizer.convert_tokens_to_ids("[CLS]")`
- Encode plus: `encoding = tokenizer.encode_plus(text, add_special_tokens=True)`

Inference

- Forward pass: `outputs = model(**tokens)`
- Get logits: `logits = outputs.logits`
- Get predictions: `predictions = torch.argmax(logits, dim=-1)`
- Get probabilities: `probs = torch.softmax(logits, dim=-1)`
- Get hidden states: `hidden_states = outputs.hidden_states`
- Get attention: `attentions = outputs.attentions`
- Generate text: `output_ids = model.generate(input_ids, max_length=100)`
- Generate with sampling: `output_ids = model.generate(input_ids, do_sample=True, temperature=0.7)`
- Generate with beam search: `output_ids = model.generate(input_ids, num_beams=5)`
- Generate with constraints: `output_ids = model.generate(input_ids, max_new_tokens=50, no_repeat_ngram_size=2)`

Fine-tuning

- Prepare dataset: `dataset = Dataset.from_pandas(df)`
- Tokenize dataset: `tokenized = dataset.map(lambda x: tokenizer(x["text"], truncation=True), batched=True)`
- Training arguments: `args = TrainingArguments(output_dir="./results", num_train_epochs=3, per_device_train_batch_size=16)`
- Create trainer: `trainer = Trainer(model=model, args=args, train_dataset=train_data, eval_dataset=val_data)`
- Train model: `trainer.train()`
- Evaluate model: `trainer.evaluate()`
- Save model: `model.save_pretrained("./my_model")`
- Save tokenizer: `tokenizer.save_pretrained("./my_model")`
- Push to hub: `model.push_to_hub("my-model")`
- Load from hub: `model = AutoModel.from_pretrained("username/my-model")`

Pipelines

- Text classification: `classifier = pipeline("text-classification")`
- Sentiment analysis: `sentiment = pipeline("sentiment-analysis")`
- NER: `ner = pipeline("ner", aggregation_strategy="simple")`
- Question answering: `qa = pipeline("question-answering")`
- Summarization: `summarizer = pipeline("summarization")`
- Translation: `translator = pipeline("translation_en_to_fr")`

By: Waleed Mousa

- Text generation: `generator = pipeline("text-generation")`
- Fill mask: `fill_mask = pipeline("fill-mask")`
- Zero-shot classification: `classifier = pipeline("zero-shot-classification")`
- Feature extraction: `extractor = pipeline("feature-extraction")`

📊 MLFLOW EXPERIMENT TRACKING

Experiment Setup

- Set tracking URI: `mlflow.set_tracking_uri("http://localhost:5000")`
- Set experiment: `mlflow.set_experiment("my-experiment")`
- Create experiment: `experiment_id = mlflow.create_experiment("my-experiment")`
- Get experiment: `experiment = mlflow.get_experiment_by_name("my-experiment")`
- List experiments: `experiments = mlflow.search_experiments()`
- Start run: `with mlflow.start_run(): ...`
- Start named run: `with mlflow.start_run(run_name="my-run"): ...`
- Nested run: `with mlflow.start_run(nested=True): ...`
- End run: `mlflow.end_run()`
- Get active run: `run = mlflow.active_run()`

Logging Parameters

- Log parameter: `mlflow.log_param("learning_rate", 0.01)`
- Log parameters dict: `mlflow.log_params({"lr": 0.01, "batch_size": 32})`
- Log metric: `mlflow.log_metric("accuracy", 0.95)`
- Log metric with step: `mlflow.log_metric("loss", 0.5, step=epoch)`
- Log metrics dict: `mlflow.log_metrics({"acc": 0.95, "f1": 0.92})`
- Log artifact: `mlflow.log_artifact("model.pkl")`
- Log artifacts dir: `mlflow.log_artifacts("./outputs")`
- Log figure: `mlflow.log_figure(fig, "plot.png")`
- Log text: `mlflow.log_text("description", "notes.txt")`
- Log dict: `mlflow.log_dict(config, "config.json")`

Model Logging

- Log sklearn model: `mlflow.sklearn.log_model(model, "model")`
- Log pytorch model: `mlflow.pytorch.log_model(model, "model")`
- Log tensorflow model: `mlflow.tensorflow.log_model(model, "model")`
- Log model with signature: `mlflow.sklearn.log_model(model, "model", signature=signature)`
- Infer signature: `signature = mlflow.models.infer_signature(X_train, model.predict(X_train))`
- Log model with input example: `mlflow.sklearn.log_model(model, "model", input_example=X_train[:5])`
- Register model: `mlflow.register_model("runs:/run_id/model", "MyModel")`
- Load model: `model = mlflow.sklearn.load_model("runs:/run_id/model")`
- Load registered model: `model = mlflow.pyfunc.load_model("models:/MyModel/1")`

By: Waleed Mousa

- Serve model: `mlflow models serve -m "models:/MyModel/1" -p 5001`

Autologging

- Autolog sklearn: `mlflow.sklearn.autolog()`
- Autolog pytorch: `mlflow.pytorch.autolog()`
- Autolog tensorflow: `mlflow.tensorflow.autolog()`
- Autolog xgboost: `mlflow.xgboost.autolog()`
- Autolog lightgbm: `mlflow.lightgbm.autolog()`
- Autolog all: `mlflow.autolog()`
- Disable autolog: `mlflow.autolog(disable=True)`
- Autolog with options: `mlflow.sklearn.autolog(log_models=True, log_input_examples=True)`
- Exclusive autolog: `mlflow.sklearn.autolog(exclusive=True)`
- Silent autolog: `mlflow.sklearn.autolog(silent=True)`

Querying Runs

- Search runs: `runs = mlflow.search_runs(experiment_ids=["1"])`
- Filter runs: `runs = mlflow.search_runs(filter_string="metrics.accuracy > 0.9")`
- Order runs: `runs = mlflow.search_runs(order_by=["metrics.accuracy DESC"])`
- Get run: `run = mlflow.get_run(run_id)`
- Get run params: `params = run.data.params`
- Get run metrics: `metrics = run.data.metrics`
- Get artifact URI: `artifact_uri = run.info.artifact_uri`
- Download artifacts: `mlflow.artifacts.download_artifacts(run_id=run_id, dst_path="./downloads")`
- Compare runs: `mlflow.search_runs().sort_values("metrics.accuracy", ascending=False)`
- Delete run: `mlflow.delete_run(run_id)`

🐳 DOCKER FOR ML

Dockerfile Basics

- Base image: `FROM python:3.10-slim`
- Set workdir: `WORKDIR /app`
- Copy requirements: `COPY requirements.txt .`
- Install dependencies: `RUN pip install --no-cache-dir -r requirements.txt`
- Copy source: `COPY . .`
- Expose port: `EXPOSE 8000`
- Set entrypoint: `ENTRYPOINT ["python", "app.py"]`
- Set command: `CMD ["--port", "8000"]`
- Set environment: `ENV MODEL_PATH=/app/model`
- Add label: `LABEL maintainer="name@email.com"`

Docker Commands

- Build image: `docker build -t my-ml-app .`
- Build with args: `docker build --build-arg VERSION=1.0 -t my-ml-app .`
- Run container: `docker run -p 8000:8000 my-ml-app`

By: Waleed Mousa

- Run detached: `docker run -d -p 8000:8000 my-ml-app`
- Run with volume: `docker run -v $(pwd)/data:/app/data my-ml-app`
- Run with GPU: `docker run --gpus all my-ml-app`
- Run with env vars: `docker run -e MODEL_NAME=bert my-ml-app`
- List containers: `docker ps -a`
- Stop container: `docker stop container_id`
- Remove container: `docker rm container_id`
- View logs: `docker logs container_id`
- Execute in container: `docker exec -it container_id bash`
- Push to registry: `docker push username/my-ml-app:v1`
- Pull from registry: `docker pull username/my-ml-app:v1`

Docker Compose

- Define service: `services: ml-api: build: . ports: - "8000:8000"`
- Add volumes: `volumes: - ./data:/app/data`
- Add environment: `environment: - MODEL_PATH=/app/model`
- Add dependencies: `depends_on: - redis`
- Start services: `docker-compose up`
- Start detached: `docker-compose up -d`
- Stop services: `docker-compose down`
- Rebuild services: `docker-compose up --build`
- Scale service: `docker-compose up --scale worker=3`
- View logs: `docker-compose logs -f`

☁️ AWS FOR ML

S3 Operations

- Create S3 client: `s3 = boto3.client("s3")`
- Upload file: `s3.upload_file("local.csv", "bucket", "data/file.csv")`
- Download file: `s3.download_file("bucket", "data/file.csv", "local.csv")`
- List objects: `response = s3.list_objects_v2(Bucket="bucket", Prefix="data/")`
- Read CSV from S3: `df = pd.read_csv("s3://bucket/data/file.csv")`
- Write CSV to S3: `df.to_csv("s3://bucket/data/file.csv", index=False)`
- Delete object: `s3.delete_object(Bucket="bucket", Key="data/file.csv")`
- Copy object: `s3.copy_object(CopySource="bucket/src", Bucket="bucket", Key="dst")`
- Generate presigned URL: `url = s3.generate_presigned_url("get_object", Params={"Bucket": "bucket", "Key": "file"}, ExpiresIn=3600)`
- Upload with transfer: `from boto3.s3.transfer import TransferConfig; s3.upload_file("file", "bucket", "key", Config=TransferConfig(multipart_threshold=1024*25))`

SageMaker

- Create session: `session = sagemaker.Session()`
- Get execution role: `role = sagemaker.get_execution_role()`

By: Waleed Mousa

- Create estimator: `estimator = Estimator(image_uri, role, instance_count=1, instance_type="ml.m5.xlarge")`
- Train model: `estimator.fit({"train": "s3://bucket/train", "test": "s3://bucket/test"})`
- Deploy model: `predictor = estimator.deploy(instance_type="ml.t2.medium", initial_instance_count=1)`
- Make prediction: `response = predictor.predict(data)`
- Delete endpoint: `predictor.delete_endpoint()`
- Create sklearn estimator: `estimator = SKLearn(entry_point="train.py", role=role, instance_type="ml.m5.xlarge", framework_version="1.0-1")`
- Create pytorch estimator: `estimator = PyTorch(entry_point="train.py", role=role, instance_type="ml.p3.2xlarge", framework_version="1.12")`
- Hyperparameter tuning: `tuner = HyperparameterTuner(estimator, objective_metric, hyperparameter_ranges, max_jobs=20)`

EC2 & Lambda

- Create EC2 client: `ec2 = boto3.client("ec2")`
- Launch instance: `response = ec2.run_instances(ImageId="ami-xxx", InstanceType="p3.2xlarge", MinCount=1, MaxCount=1)`
- Stop instance: `ec2.stop_instances(InstanceIds=["i-xxx"])`
- Terminate instance: `ec2.terminate_instances(InstanceIds=["i-xxx"])`
- Create Lambda client: `lambda_client = boto3.client("lambda")`
- Invoke Lambda: `response = lambda_client.invoke(FunctionName="my-function", Payload=json.dumps(event))`
- Create function: `lambda_client.create_function(FunctionName="my-function", Runtime="python3.9", Handler="lambda_function.handler", Code={"ZipFile": zip_bytes}, Role=role_arn)`
- Update function code: `lambda_client.update_function_code(FunctionName="my-function", ZipFile=zip_bytes)`
- Add layer: `lambda_client.update_function_configuration(FunctionName="my-function", Layers=["arn:aws:lambda:region:xxx:layer:name:1"])`
- Set environment: `lambda_client.update_function_configuration(FunctionName="my-function", Environment={"Variables": {"KEY": "value"}})`

📈 MODEL DEPLOYMENT

FastAPI Serving

- Create app: `app = FastAPI()`
- Health endpoint: `@app.get("/health") def health(): return {"status": "healthy"}`
- Prediction endpoint: `@app.post("/predict") def predict(data: InputData): return model.predict(data)`

By: Waleed Mousa

- Load model on startup: @app.on_event("startup") def load_model(): global model; model = joblib.load("model.pkl")
- Add CORS: app.add_middleware(CORSMiddleware, allow_origins=["*"])
- Pydantic model: class InputData(BaseModel): feature1: float; feature2: float
- Async endpoint: @app.post("/predict") async def predict(data: InputData): return await model.predict(data)
- Background task: @app.post("/train") def train(background_tasks: BackgroundTasks): background_tasks.add_task(train_model)
- File upload: @app.post("/upload") def upload(file: UploadFile): return {"filename": file.filename}
- Run server: uvicorn.run(app, host="0.0.0.0", port=8000)

Flask Serving

- Create app: app = Flask(__name__)
- Prediction endpoint: @app.route("/predict", methods=["POST"]) def predict(): return jsonify(model.predict(request.json))
- Load model: model = joblib.load("model.pkl")
- Get JSON data: data = request.get_json()
- Return JSON: return jsonify({"prediction": pred})
- Error handling: @app.errorhandler(500) def error(e): return jsonify({"error": str(e)}), 500
- Run app: app.run(host="0.0.0.0", port=5000)
- Production server: gunicorn -w 4 -b 0.0.0.0:5000 app:app
- With gevent: gunicorn -w 4 -k gevent -b 0.0.0.0:5000 app:app
- Debug mode: app.run(debug=True)

Model Serialization

- Save with joblib: joblib.dump(model, "model.joblib")
- Load with joblib: model = joblib.load("model.joblib")
- Save with pickle: pickle.dump(model, open("model.pkl", "wb"))
- Load with pickle: model = pickle.load(open("model.pkl", "rb"))
- Save PyTorch: torch.save(model.state_dict(), "model.pth")
- Load PyTorch: model.load_state_dict(torch.load("model.pth"))
- Save ONNX: torch.onnx.export(model, dummy_input, "model.onnx")
- Load ONNX: import onnxruntime; session = onnxruntime.InferenceSession("model.onnx")
- Save TensorFlow: model.save("model_dir")
- Load TensorFlow: model = tf.keras.models.load_model("model_dir")

Batch Inference

- Predict batch: predictions = model.predict(X_batch)
- Chunk data: chunks = [data[i:i+batch_size] for i in range(0, len(data), batch_size)]
- Process chunks: results = [model.predict(chunk) for chunk in chunks]
- Parallel processing: from joblib import Parallel, delayed; results = Parallel(n_jobs=-1)(delayed(model.predict)(chunk) for chunk in chunks)

By: Waleed Mousa

- Async batch: `results = await asyncio.gather(*[predict(chunk) for chunk in chunks])`
- Generator batches: `def batch_generator(data, size): for i in range(0, len(data), size): yield data[i:i+size]`
- Memory efficient: `for batch in batch_generator(data, 1000): process(batch)`
- Progress bar: `from tqdm import tqdm; results = [model.predict(chunk) for chunk in tqdm(chunks)]`
- Save results: `pd.DataFrame(results).to_csv("predictions.csv")`
- Stream results: `for pred in model.predict_generator(data): yield pred`

🔧 DATA VALIDATION

Great Expectations

- Create context: `context = gx.get_context()`
- Add datasource: `datasource = context.sources.add_pandas("my_datasource")`
- Add data asset: `asset = datasource.add_dataframe_asset("my_asset")`
- Build batch request: `batch_request = asset.build_batch_request(dataframe=df)`
- Create expectation: `validator.expect_column_values_to_not_be_null("column")`
- Expect unique: `validator.expect_column_values_to_be_unique("id")`
- Expect in set: `validator.expect_column_values_to_be_in_set("status", ["A", "B", "C"])`
- Expect between: `validator.expect_column_values_to_be_between("age", 0, 120)`
- Expect regex: `validator.expect_column_values_to_match_regex("email", r"^[\w.-]+@[\w.-]+\.\w+$")`
- Run validation: `results = validator.validate()`

Pydantic Validation

- Define model: `class Data(BaseModel): name: str; age: int; email: EmailStr`
- Optional field: `class Data(BaseModel): name: str; nickname: Optional[str] = None`
- Constrained field: `class Data(BaseModel): age: int = Field(ge=0, le=120)`
- Validate: `data = Data(name="John", age=30, email="john@email.com")`
- Validation error: `try: Data(**input) except ValidationError as e: print(e.json())`
- Custom validator: `@validator("age") def check_age(cls, v): assert v >= 0; return v`
- Root validator: `@root_validator def check(cls, values): assert values["end"] > values["start"]; return values`
- Config: `class Config: extra = "forbid"`
- From dict: `data = Data.parse_obj({"name": "John", "age": 30})`
- To dict: `data.dict()`

📚 QUICK REFERENCE CARD

Essential Imports

By: Waleed Mousa

```python
# Data manipulation
import pandas as pd
import numpy as np

# ML libraries
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder
from sklearn.metrics import accuracy_score, f1_score, mean_squared_error, r2_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression, LinearRegression

# Deep learning
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset

# Transformers
from transformers import AutoModel, AutoTokenizer, Trainer, TrainingArguments

# Experiment tracking
import mlflow
import mlflow.sklearn

# Deployment
from fastapi import FastAPI
import joblib
```

ML Pipeline Template

```python
# 1. Load data
df = pd.read_csv("data.csv")

# 2. Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# 3. Preprocess
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 4. Train model
model = RandomForestClassifier()
model.fit(X_train_scaled, y_train)

# 5. Evaluate
y_pred = model.predict(X_test_scaled)
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
```

By: Waleed Mousa

```
# 6. Save model
joblib.dump(model, "model.joblib")
```

PyTorch Training Template

```
# Training loop
for epoch in range(num_epochs):
    model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        outputs = model(batch["input"])
        loss = criterion(outputs, batch["target"])
        loss.backward()
        optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_loss = sum(criterion(model(b["input"]), b["target"]) for b in
val_loader)
```

Common Commands

```
# Start MLflow UI
mlflow ui --port 5000

# Start FastAPI server
uvicorn app:app --reload --host 0.0.0.0 --port 8000

# Build Docker image
docker build -t ml-app .

# Run with GPU
docker run --gpus all -p 8000:8000 ml-app

# Train on SageMaker
python -c "from sagemaker import get_execution_role; print(get_execution_role())"
```