Scikit-learn Intermediate Cheat Sheet

1. Pipelines: Chaining Steps

Pipelines are the most important tool for intermediate ML. They chain preprocessing and modeling steps into a single object, preventing data leakage.

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler,
    OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression

# Create a sample DataFrame with mixed data types
# X = pd.DataFrame(...)

# 1. Define preprocessing for numeric columns
numeric_features = ['age', 'salary']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

# 2. Define preprocessing for categorical columns
categorical_features = ['city', 'gender']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='
    most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'
    ))])

# 3. Combine preprocessing steps
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer,
    categorical_features)])

# 4. Create the final pipeline with a model
model_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression())])

# Now, use the pipeline as a single estimator
# model_pipeline.fit(X_train, y_train)
# model_pipeline.predict(X_test)
```

2. Cross-Validation

A more reliable way to evaluate model performance than a single train-test split. `k-fold` cross-validation splits the data into 'k' folds, training on 'k-1' and testing on 1, and repeats 'k' times.

```
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

# Create a model
knn = KNeighborsClassifier(n_neighbors=5)

# Perform 5-fold cross-validation
# (using raw iris data for simplicity)
scores = cross_val_score(knn, X, y, cv=5, scoring='
    accuracy')

print(f'Scores for each fold: {scores}')
print(f'Mean accuracy: {scores.mean():.2f}')
print(f'Std deviation: {scores.std():.2f}')
```

3. Hyperparameter Tuning

Find the best settings for your model automatically.

Grid Search ('GridSearchCV')

Exhaustively tries every combination of parameters you specify.

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Define the parameter grid
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5]
}

# Instantiate GridSearchCV
grid_search = GridSearchCV(
    estimator=RandomForestClassifier(random_state
    =42),
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1 # Use all available CPU cores
)

# Fit it to the data
# grid_search.fit(X_train, y_train)
```

For large parameter spaces, use `RandomizedSearchCV` to sample a fixed number of combinations.

4. Common Intermediate Models

Support Vector Machines (SVM)

Finds the optimal hyperplane that separates classes. Powerful, but can be slow on large datasets.

```
from sklearn.svm import SVC
svc_model = SVC(kernel='rbf', C=1.0, gamma='scale')
```

Random Forest

An ensemble of decision trees. Reduces overfitting and is generally robust.

```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier(n_estimators=100)
```

Gradient Boosting

Builds trees sequentially, where each tree corrects the errors of the previous one. Often gives the best performance.

```
from sklearn.ensemble import
    GradientBoostingClassifier
gb_model = GradientBoostingClassifier(n_estimators
    =100)
```

5. Advanced Classification Metrics

Precision, Recall, F1-Score

Crucial for imbalanced datasets where accuracy is misleading.

- **Precision**: Of all positive predictions, how many were correct? (TP / (TP + FP))
- **Recall**: Of all actual positive instances, how many did the model find? (TP / (TP + FN))

- **F1-Score**: Harmonic mean of precision and recall.

```
from sklearn.metrics import classification_report
# y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

ROC Curve and AUC

Visualizes the trade-off between the true positive rate and false positive rate. AUC (Area Under Curve) is a single number summarizing this performance. An AUC of 1.0 is perfect; 0.5 is random.

```
from sklearn.metrics import RocCurveDisplay
# model.fit(X_train, y_train)
RocCurveDisplay.from_estimator(model, X_test, y_test
    )
plt.show()
```

6. Principal Component Analysis (PCA)

An unsupervised technique to reduce the number of features in a dataset while retaining as much variance as possible. Often used for visualization or as a preprocessing step.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2) # Reduce to 2 dimensions

# Fit and transform the data
X_reduced = pca.fit_transform(X_scaled)

# Check how much variance is explained by the
    components
print(pca.explained_variance_ratio_)
print(f'Total explained: {pca.
    explained_variance_ratio_.sum():.2f}')
```

7. Handling Imbalanced Data

When one class is much more frequent than another.

Adjusting Class Weights

Many models have a `class_weight` parameter that automatically penalizes mistakes on the minority class more heavily.

```
# Automatically adjust weights inversely
    proportional to class frequencies
model = LogisticRegression(class_weight='balanced')
```

Another popular technique is resampling (e.g., oversampling the minority class with SMOTE from the `imbalanced-learn` library).

8. Feature Importance

Tree-based models can provide insights into which features were most important for making predictions.

```
# Assume 'rf_model' is a trained RandomForest
importances = rf_model.feature_importances_
feature_names = X.columns # If X is a DataFrame

# Create a series for easier viewing
feat_imp = pd.Series(importances, index=
    feature_names)
feat_imp.nlargest(10).plot(kind='barh')
```