# Machine Learning Foundations - Intermediate Handout

Machine Learning for Smarter Innovation

## 1 Machine Learning Foundations - Intermediate Handout

**Target Audience**: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

---

### 1.1 Setup and Environment

Before implementing machine learning workflows, ensure you have the required libraries installed. This handout uses scikit-learn as the primary ML library, with pandas for data manipulation and matplotlib for visualization.

```
# Core data science libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn: data preparation
from sklearn.model_selection import train_test_split, cross_val_score,
    GridSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder
from sklearn.impute import SimpleImputer

# Scikit-learn: models
from sklearn.linear_model import LogisticRegression, LinearRegression, Ridge
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

# Scikit-learn: evaluation
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
    f1_score,
                             confusion_matrix, classification_report,
                             mean_squared_error, r2_score, silhouette_score)

# Scikit-learn: pipelines
from sklearn.pipeline import Pipeline

# Set random seed for reproducibility
np.random.seed(42)

# Configure plotting
```

```
plt.rcParams['figure.figsize'] = (10, 6)
sns.set_style('whitegrid')
```

---

## 1.2 1. The Machine Learning Workflow

### 1.2.1 Concept Overview

Every ML project follows a consistent workflow regardless of the specific algorithm or problem type. Understanding this workflow helps you organize projects systematically and avoid common pitfalls. The workflow has five main phases: data preparation, model selection, training, evaluation, and deployment.

### 1.2.2 Implementation

```
class MLWorkflow:
    """
    A structured approach to machine learning projects.
    This class encapsulates the standard ML workflow.
    """

    def __init__(self, random_state=42):
        self.random_state = random_state
        self.scaler = None
        self.model = None
        self.feature_names = None

    def prepare_data(self, df, target_column, test_size=0.2):
        """
        Phase 1: Data Preparation

        Handles missing values, encodes categories, scales features,
        and splits into train/test sets.
        """
        # Separate features and target
        X = df.drop(columns=[target_column]).copy()
        y = df[target_column].copy()
        self.feature_names = X.columns.tolist()

        # Identify column types
        numeric_cols = X.select_dtypes(include=[np.number]).columns
        categorical_cols = X.select_dtypes(include=['object', 'category']).
    columns

        # Handle missing values
        if X[numeric_cols].isnull().any().any():
            imputer = SimpleImputer(strategy='median')
            X[numeric_cols] = imputer.fit_transform(X[numeric_cols])

        if X[categorical_cols].isnull().any().any():
            for col in categorical_cols:
                X[col] = X[col].fillna(X[col].mode()[0])

        # Encode categorical variables
        for col in categorical_cols:
            le = LabelEncoder()
            X[col] = le.fit_transform(X[col].astype(str))

        # Split data
```

```
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=test_size, random_state=self.random_state
        )

        # Scale features
        self.scaler = StandardScaler()
        X_train_scaled = pd.DataFrame(
            self.scaler.fit_transform(X_train),
            columns=X_train.columns,
            index=X_train.index
        )
        X_test_scaled = pd.DataFrame(
            self.scaler.transform(X_test),
            columns=X_test.columns,
            index=X_test.index
        )

        print(f"Training samples: {len(X_train)}")
        print(f"Test samples: {len(X_test)}")
        print(f"Features: {X_train.shape[1]}")

        return X_train_scaled, X_test_scaled, y_train, y_test

    def select_model(self, X_train, y_train, X_test, y_test, task='
classification'):
        """
        Phase 2: Model Selection

        Compares multiple algorithms to find the best performer.
        """
        if task == 'classification':
            models = {
                'Logistic Regression': LogisticRegression(max_iter=1000,
random_state=self.random_state),
                'Decision Tree': DecisionTreeClassifier(max_depth=10,
random_state=self.random_state),
                'Random Forest': RandomForestClassifier(n_estimators=100,
random_state=self.random_state),
                'Gradient Boosting': GradientBoostingClassifier(n_estimators
=100, random_state=self.random_state),
                'KNN': KNeighborsClassifier(n_neighbors=5)
            }
            scoring = 'accuracy'
        else:
            models = {
                'Linear Regression': LinearRegression(),
                'Ridge Regression': Ridge(alpha=1.0),
                'Decision Tree': DecisionTreeRegressor(max_depth=10,
random_state=self.random_state),
                'Random Forest': RandomForestRegressor(n_estimators=100,
random_state=self.random_state)
            }
            scoring = 'r2'

        results = []
        for name, model in models.items():
            # Cross-validation on training data
            cv_scores = cross_val_score(model, X_train, y_train, cv=5, scoring
=scoring)

            # Fit and evaluate on test
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
```

```
            if task == 'classification':
                test_score = accuracy_score(y_test, y_pred)
            else:
                test_score = r2_score(y_test, y_pred)

            results.append({
                'Model': name,
                'CV Mean': cv_scores.mean(),
                'CV Std': cv_scores.std(),
                'Test Score': test_score
            })

        results_df = pd.DataFrame(results).sort_values('Test Score', ascending
    =False)
        print("\nModel Comparison:")
        print(results_df.to_string(index=False))

        # Select best model
        best_name = results_df.iloc[0]['Model']
        self.model = models[best_name]
        print(f"\nSelected model: {best_name}")

        return results_df

    def train(self, X_train, y_train):
        """
        Phase 3: Model Training

        Fits the selected model on training data.
        """
        if self.model is None:
            raise ValueError("No model selected. Run select_model first.")

        self.model.fit(X_train, y_train)
        print("Model trained successfully.")
        return self.model

    def evaluate(self, X_test, y_test, task='classification'):
        """
        Phase 4: Model Evaluation

        Comprehensive evaluation on held-out test data.
        """
        y_pred = self.model.predict(X_test)

        if task == 'classification':
            print("\n" + "="*50)
            print("CLASSIFICATION REPORT")
            print("="*50)
            print(classification_report(y_test, y_pred))

            # Confusion matrix
            cm = confusion_matrix(y_test, y_pred)
            plt.figure(figsize=(8, 6))
            sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
            plt.xlabel('Predicted')
            plt.ylabel('Actual')
            plt.title('Confusion Matrix')
            plt.tight_layout()
            plt.savefig('confusion_matrix.png', dpi=150)
            plt.close()
            print("Confusion matrix saved to confusion_matrix.png")
```

```
        else:
            rmse = np.sqrt(mean_squared_error(y_test, y_pred))
            r2 = r2_score(y_test, y_pred)

            print("\n" + "="*50)
            print("REGRESSION METRICS")
            print("="*50)
            print(f"RMSE: {rmse:.4f}")
            print(f"R-squared: {r2:.4f}")

            # Actual vs Predicted plot
            plt.figure(figsize=(8, 6))
            plt.scatter(y_test, y_pred, alpha=0.5)
            plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()
],
                    'r--', lw=2)
            plt.xlabel('Actual')
            plt.ylabel('Predicted')
            plt.title('Actual vs Predicted')
            plt.tight_layout()
            plt.savefig('actual_vs_predicted.png', dpi=150)
            plt.close()
            print("Prediction plot saved to actual_vs_predicted.png")

        return y_pred

    def get_feature_importance(self, top_n=10):
        """
        Extract and visualize feature importance.
        """
        if hasattr(self.model, 'feature_importances_'):
            importance = self.model.feature_importances_
        elif hasattr(self.model, 'coef_'):
            importance = np.abs(self.model.coef_).flatten()
        else:
            print("Model does not support feature importance.")
            return None

        importance_df = pd.DataFrame({
            'Feature': self.feature_names,
            'Importance': importance
        }).sort_values('Importance', ascending=False).head(top_n)

        plt.figure(figsize=(10, 6))
        plt.barh(importance_df['Feature'][::-1], importance_df['Importance'
][::-1])
        plt.xlabel('Importance')
        plt.title(f'Top {top_n} Feature Importances')
        plt.tight_layout()
        plt.savefig('feature_importance.png', dpi=150)
        plt.close()

        print(f"\nTop {top_n} Features:")
        print(importance_df.to_string(index=False))

        return importance_df
```

### 1.2.3 Usage Example

```
# Load sample data
```

```
from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

# Run complete workflow
workflow = MLWorkflow(random_state=42)

# Phase 1: Prepare data
X_train, X_test, y_train, y_test = workflow.prepare_data(df, 'target')

# Phase 2: Select best model
results = workflow.select_model(X_train, y_train, X_test, y_test)

# Phase 3: Train final model
workflow.train(X_train, y_train)

# Phase 4: Evaluate
predictions = workflow.evaluate(X_test, y_test)

# Get feature importance
importance = workflow.get_feature_importance()
```

---

## 1.3 2. Data Preprocessing Techniques

### 1.3.1 Concept Overview

Raw data almost always requires preprocessing before ML algorithms can use it effectively. The three most common preprocessing tasks are handling missing values, encoding categorical variables, and scaling numerical features. Each technique has implications for model performance.

### 1.3.2 Missing Value Handling

```
def demonstrate_missing_value_handling():
    """
    Compare different strategies for handling missing values.
    """
    # Create data with missing values
    np.random.seed(42)
    data = {
        'age': [25, 30, np.nan, 45, 50, np.nan, 35, 40],
        'income': [50000, np.nan, 75000, 100000, np.nan, 80000, 60000, 90000],
        'category': ['A', 'B', 'A', np.nan, 'B', 'A', 'B', 'A']
    }
    df = pd.DataFrame(data)

    print("Original Data:")
    print(df)
    print(f"\nMissing values:\n{df.isnull().sum()}")

    # Strategy 1: Drop missing values
    df_dropped = df.dropna()
    print(f"\nAfter dropping: {len(df_dropped)} rows remain")

    # Strategy 2: Mean/Median imputation for numerical
    imputer_mean = SimpleImputer(strategy='mean')
```

```
    imputer_median = SimpleImputer(strategy='median')

    df_mean = df.copy()
    df_mean[['age', 'income']] = imputer_mean.fit_transform(df[['age', 'income
    ']])

    df_median = df.copy()
    df_median[['age', 'income']] = imputer_median.fit_transform(df[['age', '
    income']])

    print(f"\nMean imputation - age mean: {df_mean['age'].mean():.1f}")
    print(f"Median imputation - age median: {df_median['age'].median():.1f}")

    # Strategy 3: Mode imputation for categorical
    df_mode = df.copy()
    df_mode['category'] = df_mode['category'].fillna(df_mode['category'].mode
    ()[0])
    print(f"\nMode imputation - category mode: {df_mode['category'].mode()[0]}
    ")

    return df_mean, df_median, df_mode

df_mean, df_median, df_mode = demonstrate_missing_value_handling()
```

### 1.3.3 Feature Scaling

```
def demonstrate_scaling():
    """
    Compare StandardScaler and MinMaxScaler.
    """
    # Create sample data
    np.random.seed(42)
    data = pd.DataFrame({
        'small_scale': np.random.randn(100),  # Mean 0, Std 1
        'large_scale': np.random.randn(100) * 1000 + 5000,  # Mean 5000, Std
    1000
        'binary': np.random.randint(0, 2, 100)
    })

    print("Original Statistics:")
    print(data.describe().round(2))

    # StandardScaler: zero mean, unit variance
    scaler_standard = StandardScaler()
    data_standard = pd.DataFrame(
        scaler_standard.fit_transform(data),
        columns=data.columns
    )

    print("\nAfter StandardScaler:")
    print(data_standard.describe().round(2))

    # MinMaxScaler: scale to [0, 1]
    scaler_minmax = MinMaxScaler()
    data_minmax = pd.DataFrame(
        scaler_minmax.fit_transform(data),
        columns=data.columns
    )

    print("\nAfter MinMaxScaler:")
    print(data_minmax.describe().round(2))
```

```
    # Visualization
    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    axes[0].boxplot([data['small_scale'], data['large_scale']])
    axes[0].set_xticklabels(['Small', 'Large'])
    axes[0].set_title('Original Data')

    axes[1].boxplot([data_standard['small_scale'], data_standard['large_scale']])
    axes[1].set_xticklabels(['Small', 'Large'])
    axes[1].set_title('StandardScaler')

    axes[2].boxplot([data_minmax['small_scale'], data_minmax['large_scale']])
    axes[2].set_xticklabels(['Small', 'Large'])
    axes[2].set_title('MinMaxScaler')

    plt.tight_layout()
    plt.savefig('scaling_comparison.png', dpi=150)
    plt.close()

    return data_standard, data_minmax

data_standard, data_minmax = demonstrate_scaling()
```

### 1.3.4 Categorical Encoding

```
def demonstrate_encoding():
    """
    Compare Label Encoding and One-Hot Encoding.
    """
    # Sample data
    data = pd.DataFrame({
        'color': ['red', 'blue', 'green', 'red', 'blue'],
        'size': ['S', 'M', 'L', 'M', 'S']
    })

    print("Original Data:")
    print(data)

    # Label Encoding: maps categories to integers
    le_color = LabelEncoder()
    le_size = LabelEncoder()

    data_label = data.copy()
    data_label['color'] = le_color.fit_transform(data['color'])
    data_label['size'] = le_size.fit_transform(data['size'])

    print("\nLabel Encoded:")
    print(data_label)
    print(f"Color mapping: {dict(zip(le_color.classes_, range(len(le_color.classes_))))}")

    # One-Hot Encoding: creates binary columns
    data_onehot = pd.get_dummies(data, columns=['color', 'size'])

    print("\nOne-Hot Encoded:")
    print(data_onehot)

    return data_label, data_onehot
```

```
data_label, data_onehot = demonstrate_encoding()
```

---

## 1.4 3. Cross-Validation and Model Selection

### 1.4.1 Concept Overview

A single train-test split can give misleading results depending on which examples happen to fall in each set. Cross-validation addresses this by repeatedly splitting data into different train and test sets, then averaging results. This provides more reliable performance estimates and helps detect overfitting.

### 1.4.2 Implementation

```
def comprehensive_cross_validation(X, y, models, cv=5):
    """
    Perform cross-validation across multiple models with multiple metrics.

    Parameters:
    -----------
    X : array-like
        Features
    y : array-like
        Target
    models : dict
        Dictionary of model name -> model instance
    cv : int
        Number of cross-validation folds
    """
    results = []

    for name, model in models.items():
        # Multiple scoring metrics
        accuracy_scores = cross_val_score(model, X, y, cv=cv, scoring='
    accuracy')
        f1_scores = cross_val_score(model, X, y, cv=cv, scoring='f1_weighted')

        results.append({
            'Model': name,
            'Accuracy Mean': accuracy_scores.mean(),
            'Accuracy Std': accuracy_scores.std(),
            'F1 Mean': f1_scores.mean(),
            'F1 Std': f1_scores.std()
        })

        print(f"{name}:")
        print(f"  Accuracy: {accuracy_scores.mean():.4f} (+/- {accuracy_scores
    .std():.4f})")
        print(f"  F1 Score: {f1_scores.mean():.4f} (+/- {f1_scores.std():.4f})
    ")
        print()

    return pd.DataFrame(results).sort_values('Accuracy Mean', ascending=False)


def visualize_cv_results(X, y, model, cv=5):
    """
    Visualize cross-validation results across folds.
    """
```

```
    from sklearn.model_selection import cross_validate

    cv_results = cross_validate(
        model, X, y, cv=cv,
        scoring=['accuracy', 'precision_weighted', 'recall_weighted', '
    f1_weighted'],
        return_train_score=True
    )

    # Create visualization
    metrics = ['accuracy', 'precision_weighted', 'recall_weighted', '
    f1_weighted']
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    for idx, metric in enumerate(metrics):
        ax = axes[idx // 2, idx % 2]

        train_key = f'train_{metric}'
        test_key = f'test_{metric}'

        folds = range(1, cv + 1)
        ax.plot(folds, cv_results[train_key], 'b-o', label='Train')
        ax.plot(folds, cv_results[test_key], 'r-o', label='Test')
        ax.set_xlabel('Fold')
        ax.set_ylabel('Score')
        ax.set_title(metric.replace('_', ' ').title())
        ax.legend()
        ax.set_xticks(folds)

    plt.tight_layout()
    plt.savefig('cv_results.png', dpi=150)
    plt.close()

    return cv_results


# Example usage
from sklearn.datasets import load_iris

data = load_iris()
X, y = data.data, data.target

models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(max_depth=5),
    'Random Forest': RandomForestClassifier(n_estimators=100),
    'KNN (k=5)': KNeighborsClassifier(n_neighbors=5)
}

results = comprehensive_cross_validation(X, y, models)
print("\nFinal Rankings:")
print(results.to_string(index=False))
```

---

## 1.5 4. Hyperparameter Tuning

### 1.5.1 Concept Overview

Most ML algorithms have hyperparameters - settings that control how the algorithm learns. The default values rarely produce optimal results for your specific data. Hyperparameter tuning systematically

searches for better settings using cross-validation to evaluate each combination.

### 1.5.2 Grid Search Implementation

```
def grid_search_tuning(X_train, y_train, X_test, y_test):
    """
    Comprehensive hyperparameter tuning using grid search.
    """
    # Define parameter grid
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [5, 10, 20, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4],
        'max_features': ['sqrt', 'log2', None]
    }

    rf = RandomForestClassifier(random_state=42)

    # Grid search with cross-validation
    grid_search = GridSearchCV(
        rf,
        param_grid,
        cv=5,
        scoring='f1_weighted',
        n_jobs=-1,
        verbose=1,
        return_train_score=True
    )

    print("Starting grid search...")
    grid_search.fit(X_train, y_train)

    print(f"\nBest parameters: {grid_search.best_params_}")
    print(f"Best CV score: {grid_search.best_score_:.4f}")

    # Evaluate on test set
    y_pred = grid_search.predict(X_test)
    test_f1 = f1_score(y_test, y_pred, average='weighted')
    print(f"Test F1 score: {test_f1:.4f}")

    # Results analysis
    results_df = pd.DataFrame(grid_search.cv_results_)
    top_results = results_df.nlargest(10, 'mean_test_score')[
        ['params', 'mean_test_score', 'std_test_score', 'rank_test_score']
    ]
    print("\nTop 10 Parameter Combinations:")
    print(top_results.to_string(index=False))

    return grid_search.best_estimator_, results_df


def random_search_tuning(X_train, y_train, X_test, y_test, n_iter=50):
    """
    Hyperparameter tuning using randomized search.
    More efficient than grid search for large parameter spaces.
    """
    from sklearn.model_selection import RandomizedSearchCV
    from scipy.stats import randint, uniform

    # Define parameter distributions
    param_distributions = {
```

```
        'n_estimators': randint(50, 300),
        'max_depth': [5, 10, 20, 30, None],
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10),
        'max_features': ['sqrt', 'log2', None],
        'bootstrap': [True, False]
    }

    rf = RandomForestClassifier(random_state=42)

    random_search = RandomizedSearchCV(
        rf,
        param_distributions,
        n_iter=n_iter,
        cv=5,
        scoring='f1_weighted',
        n_jobs=-1,
        random_state=42,
        verbose=1
    )

    print(f"Starting random search ({n_iter} iterations)...")
    random_search.fit(X_train, y_train)

    print(f"\nBest parameters: {random_search.best_params_}")
    print(f"Best CV score: {random_search.best_score_:.4f}")

    # Evaluate on test set
    y_pred = random_search.predict(X_test)
    test_f1 = f1_score(y_test, y_pred, average='weighted')
    print(f"Test F1 score: {test_f1:.4f}")

    return random_search.best_estimator_
```

---

## 1.6 5. Building ML Pipelines

### 1.6.1 Concept Overview

Pipelines chain preprocessing and modeling steps into a single object. This ensures preprocessing is applied consistently during training and prediction, prevents data leakage, and simplifies deployment. All transformations and the model are bundled together.

### 1.6.2 Implementation

```
def build_complete_pipeline():
    """
    Build a complete ML pipeline with preprocessing and model.
    """
    from sklearn.compose import ColumnTransformer
    from sklearn.preprocessing import OneHotEncoder

    # Define preprocessing for different column types
    numeric_features = ['feature1', 'feature2', 'feature3']
    categorical_features = ['category1', 'category2']

    numeric_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
```

```
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

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

    # Complete pipeline
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('classifier', RandomForestClassifier(random_state=42))
    ])

    return pipeline


def simple_pipeline_example():
    """
    Demonstrate a simple pipeline workflow.
    """
    from sklearn.datasets import load_breast_cancer

    # Load data
    data = load_breast_cancer()
    X, y = data.data, data.target
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

    # Create pipeline
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('pca', PCA(n_components=10)),
        ('classifier', RandomForestClassifier(n_estimators=100, random_state
    =42))
    ])

    # Train
    pipeline.fit(X_train, y_train)

    # Predict
    y_pred = pipeline.predict(X_test)

    # Evaluate
    print("Pipeline Results:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
    print(f"F1 Score: {f1_score(y_test, y_pred):.4f}")

    # Cross-validation on pipeline
    cv_scores = cross_val_score(pipeline, X, y, cv=5, scoring='accuracy')
    print(f"CV Accuracy: {cv_scores.mean():.4f} (+/- {cv_scores.std():.4f})")

    return pipeline

pipeline = simple_pipeline_example()
```

## 1.7 6. Model Persistence

### 1.7.1 Concept Overview

Trained models need to be saved for later use. Joblib is the recommended method for scikit-learn models because it handles NumPy arrays efficiently. Saved models can be loaded in production systems to make predictions without retraining.

### 1.7.2 Implementation

```
import joblib
import os
from datetime import datetime

def save_model(model, model_name, metadata=None):
    """
    Save trained model with metadata.
    """
    # Create models directory
    os.makedirs('models', exist_ok=True)

    # Generate filename with timestamp
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    filename = f'models/{model_name}_{timestamp}.pkl'

    # Save model and metadata
    save_data = {
        'model': model,
        'timestamp': timestamp,
        'metadata': metadata or {}
    }

    joblib.dump(save_data, filename)
    print(f"Model saved to: {filename}")

    return filename


def load_model(filename):
    """
    Load saved model and metadata.
    """
    save_data = joblib.load(filename)

    print(f"Loaded model from: {filename}")
    print(f"Saved at: {save_data['timestamp']}")

    if save_data['metadata']:
        print(f"Metadata: {save_data['metadata']}")

    return save_data['model'], save_data['metadata']


# Example usage
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

# Train a model
data = load_iris()
```

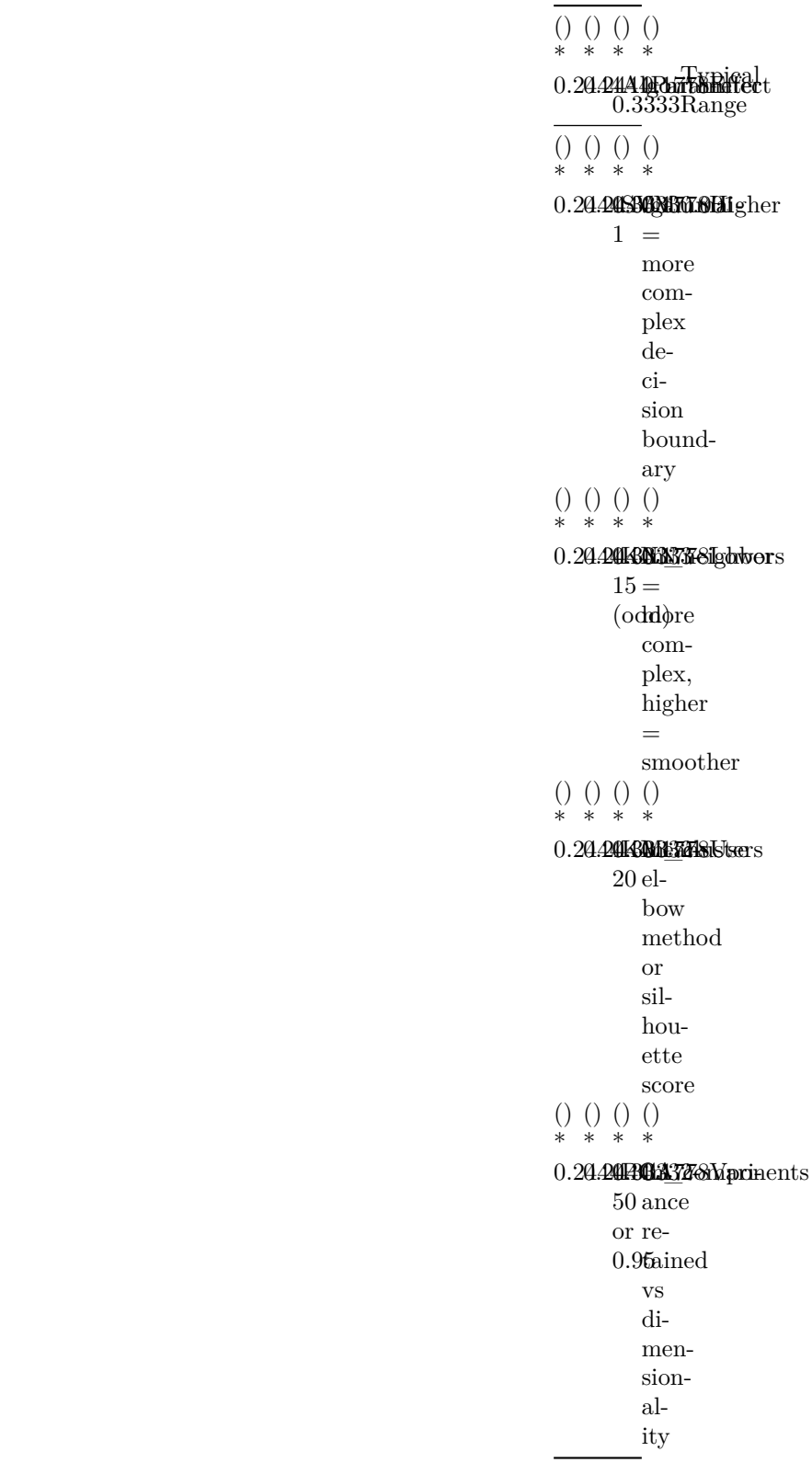
```
X, y = data.data, data.target
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X, y)

# Save with metadata
metadata = {
    'dataset': 'iris',
    'accuracy': accuracy_score(y, model.predict(X)),
    'n_features': X.shape[1],
    'n_samples': X.shape[0]
}
filename = save_model(model, 'iris_classifier', metadata)

# Load later
loaded_model, loaded_metadata = load_model(filename)
predictions = loaded_model.predict(X[:5])
print(f"Sample predictions: {predictions}")
```

---

## 1.8 Common Hyperparameters Reference

| | | | |
|---|---|---|---|
| () () () ()<br>* * * * | | | |
| 0.2 | 4 | Typical<br>0.3333 | Range |
| () () () ()<br>* * * * | | | |
| 0.2 | 4 | 500 | trees im-prove ac-cu-racy, in-crease train-ing time |
| () () () ()<br>* * * * | | | |
| 0.2 | 4 | 30 or None | trees cap-ture more com-plex-ity, risk over-fit-ting |

| | | | |
|---|---|---|---|
| () * | () * | () * | () * |
| 0.2 | 0.4 | 0.4 | Typical Range |
| () * | () * | () * | () * |
| 0.2 | 0.4 | 0.4 | Higher values prevent overfitting |
| () * | () * | () * | () * |
| 0.2 | 0.4 | 0.4 | Lower values need more estimators |
| () * | () * | () * | () * |
| 0.2 | 0.4 | 0.4 | More iterations with smaller learning rate |
| () * | () * | () * | () * |
| 0.2 | 0.4 | 0.4 | Lower C = more regularization |
| () * | () * | () * | () * |
| 0.2 | 0.4 | 0.4 | Controls margin softness |




## 1.9 Practice Projects

1. **Titanic Survival Prediction**: Classic binary classification with missing values and mixed feature types. Practice complete workflow from data cleaning to model deployment.

2. **House Price Prediction**: Regression problem with feature engineering opportunities. Experiment with polynomial features and regularization.
3. **Customer Churn**: Imbalanced classification problem. Practice class weighting and threshold tuning for business metrics.
4. **Image Classification with PCA**: Use digits dataset, reduce dimensionality with PCA, compare classifiers at different dimensionality levels.
5. **Pipeline Comparison**: Build multiple pipelines with different preprocessing strategies and compare cross-validation results systematically.

---

## 1.10 Troubleshooting

| | | |
|---|---|---|
| | | |
| * | * | * |
| 0.2727 | [illegible] | Likely |
| | 0.4242 | Cause |

| | | |
|---|---|---|
| | | |
| * | * | * |
| 0.27 | [illegible] | Try |
| train- | de- | more |
| and | fit- | com- |
| test | ting | plex |
| ac- | | model, |
| cu- | | add |
| racy | | fea- |
| | | tures |

| | | |
|---|---|---|
| | | |
| * | * | * |
| 0.27 | [illegible] | Reg- |
| train, | fit, | u- |
| low | ting | lar- |
| test | | ize, |
| ac- | | re- |
| cu- | | duce |
| racy | | fea- |
| | | tures, |
| | | get |
| | | more |
| | | data |

| | | |
|---|---|---|
| | | |
| * | * | * |
| 0.27 | [illegible] | In- |
| ver- | su- | crease |
| gen- | fit- | max_iter, |
| warn- | ing | scale |
| ing | it- | fea- |
| | er- | tures |
| | a- | |
| | tions | |

| | | |
|---|---|---|
| () | () | () |
| * | * | * |
| 0.2727 | 0.3030 | Likely Problem Cause 0.4242 |

| | | |
|---|---|---|
| () | () | () |
| * | * | * |
| 0.2727 | 0.4848 | Use a memory-efficient data, use incremental learning |

| | | |
|---|---|---|
| () | () | () |
| * | * | * |
| 0.2727 | 0.4242 | Set consistent random_state in all components |

| | | |
|---|---|---|
| () | () | () |
| * | * | * |
| 0.2727 | 0.4848 | Try different encoding strategies |

| | | |
|---|---|---|
| () | () | () |
| * | * | * |
| 0.2727 | 0.3939 | Apply scaling like StandardScaler or MinMaxScaler |


---

## 1.11 Next Steps

Ready for mathematical depth? The advanced handout covers: - Mathematical foundations of loss functions and optimization - Bias-variance tradeoff and learning curves - Regularization theory and geometric interpretation - Ensemble methods: bagging, boosting, stacking theory - Statistical testing for model comparison - Production deployment architecture

For immediate next steps: - Complete the practice projects above - Explore Kaggle competitions for real-world datasets - Study scikit-learn documentation for additional transformers - Learn about automated machine learning (AutoML) tools

---

*Implementation transforms theory into practice. Master the workflow first - data preparation, model selection, evaluation, and deployment. Then optimize within that framework. The best practitioners spend more time understanding data than tuning hyperparameters.*