1. Introduction

In this report, I'm going to walk you through a comparison of several machine learning classifiers to see how they stack up on different datasets. I've put together a lineup of models, including k-NN, SVM, Logistic Regression, LDA, QDA, MLP, and Random Forest.

I tested these models on four distinct datasets: pima_openml, breast_cancer, wine, and digits. To gauge their performance, I looked at their Accuracy, Macro F1-Score, and AUC (Area Under the Curve). I also plotted confusion matrices and ROC curves to get a better visual sense of the results.

What I found is that there's no one-size-fits-all model—different classifiers had their moments to shine on different datasets. That said, overall, I found that SVM and Random Forest delivered the most consistent and solid performance across the board.

2. Methods Implemented.

For this assignment, I implemented seven different classifiers. To keep the code modular and scalable, I broke down the entire experimental workflow into several independent stages. This covered data processing, data splitting, model training and validation, and a common base interface for all the classifiers.

2.1 Base Architecture

To standardize the interface across all models, I defined a base class called ClassifierBase, which includes fit, predict, and discriminant methods. All the classifiers I implemented, such as SVM and KNN, inherit from this base class. The discriminant method is specifically designed to return the model's decision scores or class probabilities, which is crucial for subsequent calculations of metrics like AUC.

2.2 Data Processing and Splitting

2.2.1 Data Loading

I wrote a datasets.py module to handle loading datasets from various sources in a consistent way. The load_dataset function takes a dataset name as input and fetches the corresponding data, either from scikit-learn's built-in collections (like breast_cancer, wine, and digits) or from the OpenML platform (for pima_openml). This function returns the features X, the labels y, and a metadata dictionary containing key information about the dataset, such as the number of classes and a flag for binary classification tasks.

2.2.2 Data Splitting

For model evaluation, I used the stratified_holdout function from my split.py module to partition the dataset into training and testing sets. This function was implemented with the following key features:

- **Stratification:** By setting the stratify=y parameter, I ensured that the class proportions in the training and testing sets remained consistent with the original dataset. This is especially important when working with imbalanced datasets.
- **Fixed Random Seed:** The random_state was fixed to 42. This guarantees that the exact same data split is produced every time the code is run, ensuring the reproducibility of my experimental results.
- **Fixed Split Ratio:** By default, I used an 80/20 split, allocating 80% of the data for training and the remaining 20% for testing.

2.3 Model Training and Validation

2.3.1 Preprocessing

To ensure consistency and comparability across feature scales, I applied StandardScaler to preprocess the data before training for all models, with the exception of QDA and Random Forest. This standardizer transforms each feature to have a mean of 0 and a variance of 1. The entire process was

streamlined using scikit-learn's Pipeline mechanism, which chains the feature standardization and model training steps into a single workflow.

2.3.2 5-Fold Cross-Validation

In addition to evaluating the models on a fixed test set, I also implemented 5-fold cross-validation to more robustly assess their generalization performance. This process is implemented in main.py and is triggered when the --cv 5 argument is passed. The procedure is as follows:

- The complete dataset is partitioned into 5 mutually exclusive subsets (folds) using StratifiedKFold.
- Five rounds of training and validation are performed. In each round, four of the folds are used as training data, while the remaining fold is used for validation.
- Within each round, I calculated the model's accuracy and Macro F1-Score.
- Finally, the mean and standard deviation of the results across the five rounds were computed and recorded in a _cv.csv file.

2.3.3 Overall Workflow

The overall process for model training and prediction is outlined below, illustrating each step from initial model setup to the final output. In my implementation, the main.py script is responsible for driving this entire workflow. It calls the get_models function to initialize the models, then loops through each one to perform training (fit), generate predictions (predict), and retrieve the discriminant scores (discriminant).

TrainAndPredict function

```
function:  TrainAndPredict(c, Dtr, Dte) : ⟨ŷ, s⟩
input:     c : string – classifier name.
           Dtr : ⟨Xtr, ytr⟩ – training features and labels.
           Dte : Xte – test features.
output:    ŷ : vector – class predictions for Xte.
           s : vector – discrimination scores or probabilities.
```

```
assert |Xtr| > 0 ∧ |ytr| = |Xtr|.                          // basic validation
M ← InitializeModel(c).                                    // choose model by name
Π ← CreatePipeline([Imputer(), StandardScaler(), M]).      // preprocessing + model
Π.fit(Xtr, ytr).                                           // train
ŷ ← Π.predict(Xte).                                        // hard predictions
if supports(Π, "predict_proba") then
        s ← Π.predict_proba(Xte)[:, 1].
else if supports(Π, "decision_function") then
        s ← Π.decision_function(Xte).
else
        s ← ZeroVector(|Xte|).
end-if
return ⟨ŷ, s⟩.n
```

3. Model Architectures

3.1. k-NN

I implemented the k-NN model using the KNeighborsClassifier from scikit-learn. The entire model is wrapped in a Pipeline that first standardizes the data with StandardScaler before passing it to the k-NN classifier. When predicting the class of a new sample, this model identifies the k closest neighbors in the training set and assigns the class that is most common among them as the final

prediction.

k-NN Parameter

Parameter	Value	Description
n_neighbors	5	Specifies the number of nearest neighbors to consider.
weights	"uniform"	All neighbors are weighted equally.

3.2. SVM

The goal of a Support Vector Machine (SVM) is to find an optimal hyperplane in the feature space that maximizes the margin between classes. By using the kernel trick, SVMs can also effectively handle high-dimensional and non-linear data.

For my implementation, I used scikit-learn's SVC (Support Vector Classification) module. This was also integrated with StandardScaler using a Pipeline. The discriminant method for this model returns the output of the decision_function, which represents the signed distance of a sample from the decision hyperplane.

SVM Parameter

Parameter	Value	Description
kernel	"rbf"	Uses the Radial Basis Function (RBF) as the kernel, which is suitable for handling non-linear data.
C	1.0	The regularization parameter, which balances the trade-off between a wider margin and misclassification.
gamma	"scale"	Automatically adjusts the kernel coefficient based on the number of features and the standard deviation of the data.

3.3. Logistic Regression

Logistic Regression is a linear model that uses the Sigmoid (or Logistic) function to map a linear combination of features to a probability value between 0 and 1. This probability is then used for classification. For my implementation, I used the LogisticRegression class from scikit-learn and integrated it into the Pipeline. The discriminant method for this model returns the result of predict_proba, which provides the probability of each sample belonging to each class.

Logistic Regression Parameter

Parameter	Value	Description
C	1.0	Inverse of regularization strength; smaller values specify stronger regularization.
max_iter	1000	The maximum number of iterations for the solver to converge.
solver	"lbfgs"	The algorithm used for the optimization problem; this is one of the default options in scikit-learn.

3.4. LDA & QDA

Both Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA) are generative models based on Bayes' theorem, assuming that the features follow a Gaussian distribution. LDA further assumes that all classes share the same covariance matrix, resulting in a linear decision boundary. QDA, on the other hand, relaxes this assumption by allowing each class to have its own covariance matrix, which produces a quadratic decision boundary.
For the implementation:

- **LDA**: I used scikit-learn's LinearDiscriminantAnalysis and integrated it into a Pipeline with StandardScaler.

- **QDA**: I used QuadraticDiscriminantAnalysis directly. In my implementation, StandardScaler was not applied to the QDA model.

LDA & QDA Parameter

model	Parameter	Value	Description
LDA	solver	"svd"	Uses Singular Value Decomposition, a solver that is well-suited for a high number of features as it does not require computing the covariance matrix.
QDA	reg_param	0.01	A regularization parameter that shrinks the covariance matrix, which can improve the model's stability.

3.5. MLP

A Multilayer Perceptron (MLP) is a feedforward artificial neural network composed of an input layer, one or more hidden layers, and an output layer. It is capable of learning complex, non-linear relationships in the data.

For my implementation, I used scikit-learn's MLPClassifier. To help prevent overfitting, I also included an early_stopping mechanism, which halts the training process if the validation score fails to improve after a set number of iterations.

MLP Parameter

Parameter	Value	Description
hidden_layer_sizes	(128,)	Defines a single hidden layer containing 128 neurons.
alpha	1e-4	The coefficient for the L2 regularization term, which penalizes large weights to prevent overfitting.
max_iter	500	The maximum number of training iterations.
early_stopping	True	Enables the early stopping mechanism.
n_iter_no_change	20	The number of iterations with no improvement to wait before stopping training.

3.6. Random Forest

A Random Forest is an ensemble method that operates by constructing multiple decision trees and outputting the class that is the mode of the classes from individual trees. Since each tree is trained on a random subset of both the data and the features, this approach effectively reduces the risk of overfitting.

I implemented this model using scikit-learn's RandomForestClassifier. Unlike most of the other models, my implementation does not use StandardScaler for the Random Forest, as tree-based models are not sensitive to the scale of the features.

Random Forest Parameter

Parameter	Value	Description
n_estimators	300	The number of decision trees in the forest.
max_depth	None	The maximum depth of the tree. Setting this to None means nodes are expanded until all leaves are pure.
n_jobs	-1	Uses all available CPU cores to parallelize the training process.

4. Experiments Design and Result

For my experiments, I selected four datasets. The pima_openml and breast_cancer datasets were used for binary classification tasks, while the wine and digits datasets were used for multi-class classification.

The model performance was evaluated using three primary metrics: Accuracy, Macro F1-Score, and AUC. In the following sections, I will present the performance of the different models on each of these datasets.

4.1. Pima Indians Diabetes (pima_openml)

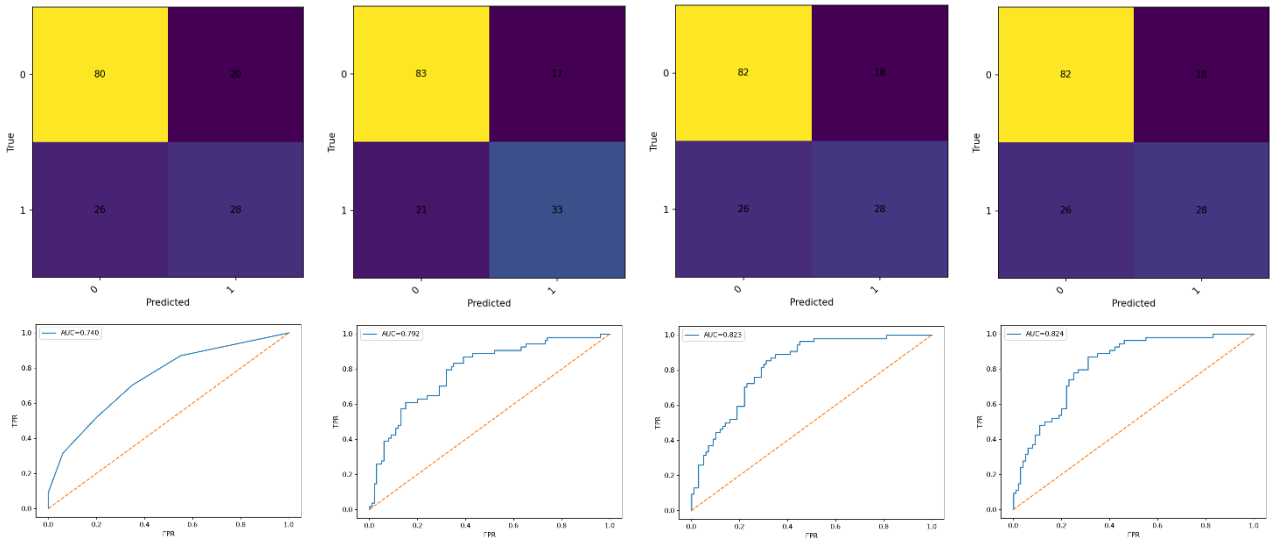
The goal of this dataset is to predict the onset of diabetes among Pima Indians.

Cross-Validation Results

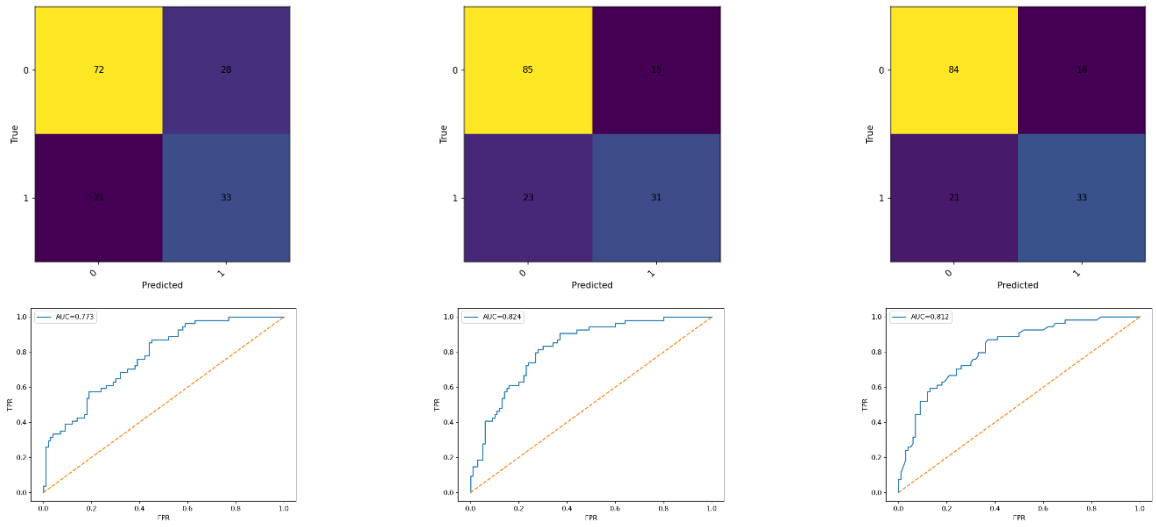
Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.728	0.762	**0.775**	0.766	0.729	0.751	0.766
Macro F1 (Mean)	0.691	0.724	**0.737**	0.727	0.695	0.709	0.733

Test Set Performance

Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.701	0.753	0.714	0.714	0.682	0.753	**0.760**
Macro F1 (Mean)	0.663	0.724	0.674	0.674	0.660	0.719	**0.730**
AUC	0.740	0.792	0.823	**0.824**	0.773	**0.824**	0.812



knn svm logreg lda



qda mlp rf

Confusion Matrices & ROC Curves for pima_openml

4.2. Breast Cancer Wisconsin (breast_cancer)

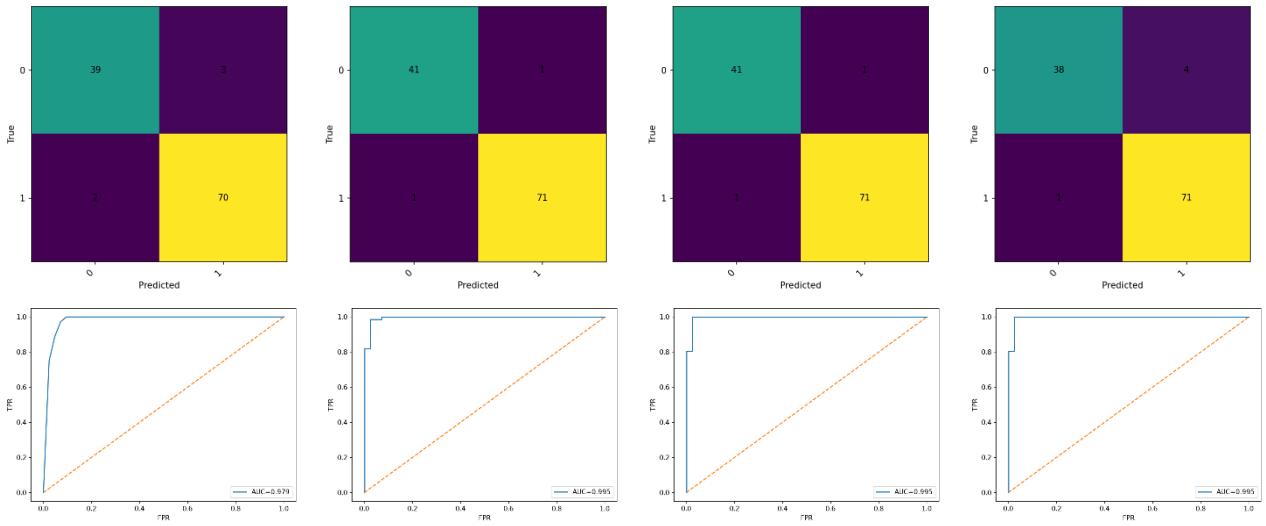
This dataset is used to predict whether a breast tumor is benign or malignant.

Cross-Validation Results

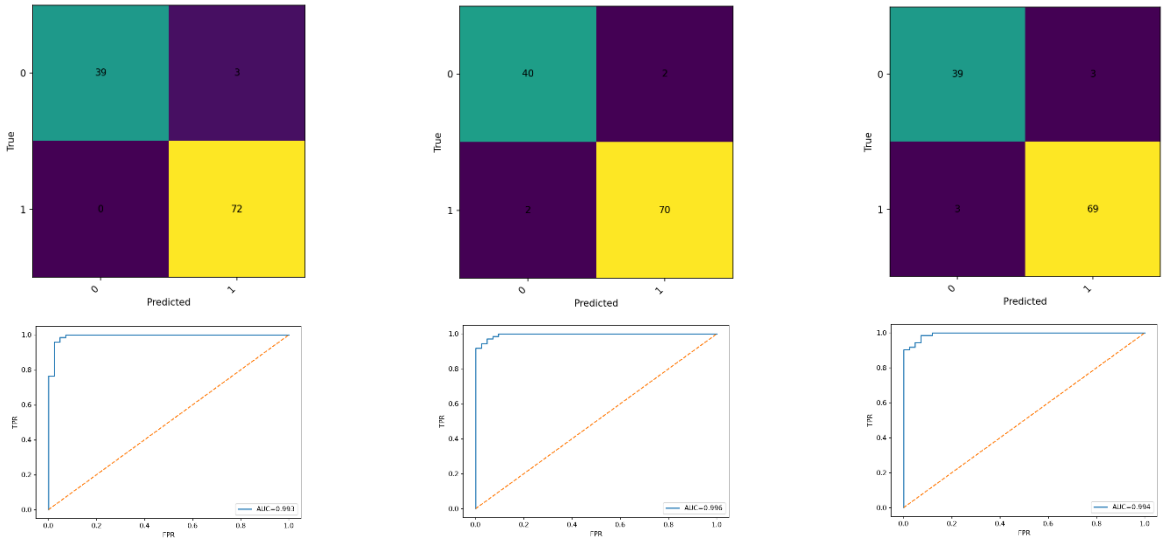
Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.963	**0.977**	0.974	0.956	0.956	0.954	0.953
Macro F1 (Mean)	0.960	**0.975**	0.971	0.952	0.952	0.951	0.949

Test Set Performance

Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.956	**0.982**	**0.982**	0.956	0.974	0.965	0.947
Macro F1 (Mean)	0.953	**0.981**	**0.981**	0.952	0.971	0.962	0.943
AUC	0.979	0.995	0.995	0.992	0.993	**0.996**	0.994



knn svm logreg lda



qda mlp rf

Confusion Matrices & ROC Curves for breast_cancer

4.3. Wine

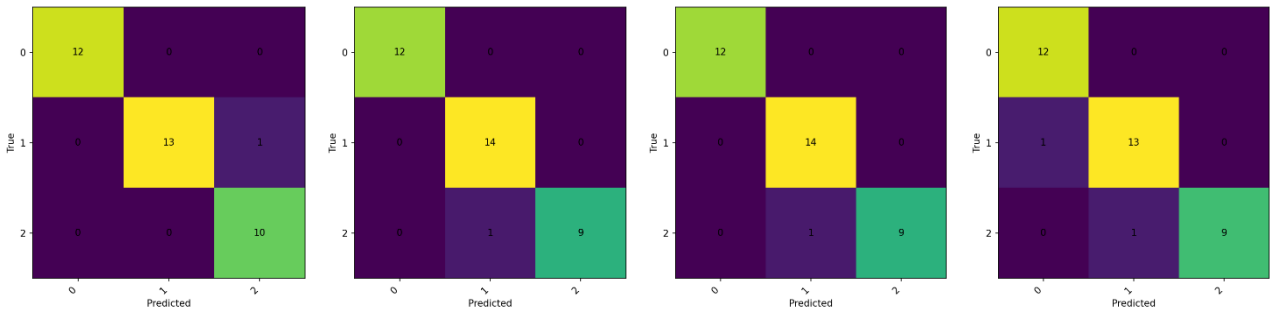
This dataset involves classifying wines into three categories based on their chemical analysis.

Cross-Validation Results

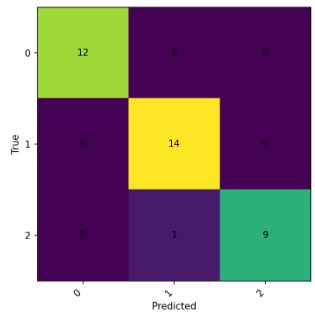
Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.972	**0.983**	**0.983**	**0.983**	**0.983**	0.820	**0.983**
Macro F1 (Mean)	0.972	0.983	0.983	0.983	**0.984**	0.817	**0.984**

Test Set Performance

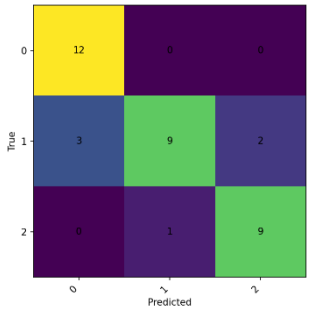
Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.972	0.972	0.972	0.944	0.972	0.833	**1.000**
Macro F1 (Mean)	0.972	0.971	0.971	0.945	0.971	0.832	**1.000**



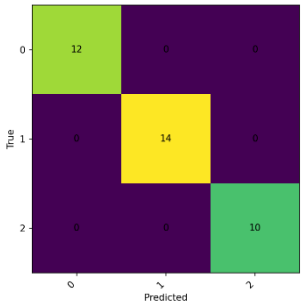
knn svm logreg lda



qda



mlp



rf

Confusion Matrices for Wine

4.4. Digits

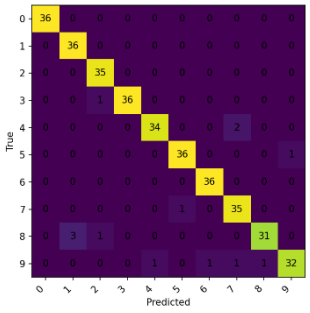
This is a handwritten digit recognition dataset, consisting of 10 classes in total.

Cross-Validation Results

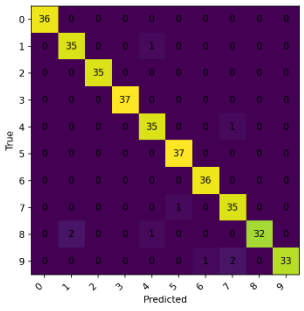
Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.979	**0.984**	0.971	0.953	0.967	0.960	0.979
Macro F1 (Mean)	0.979	**0.984**	0.971	0.953	0.967	0.960	0.979

Test Set Performance

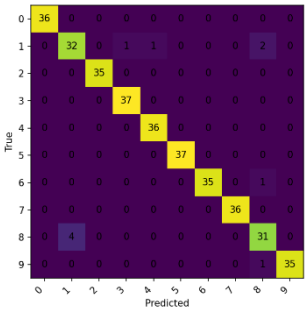
Model	k-NN	SVM	Logistic Regression	LDA	QDA	MLP	Random Forest
Accuracy (Mean)	0.964	**0.975**	0.972	0.953	0.967	0.958	0.969
Macro F1 (Mean)	0.963	**0.975**	0.972	0.952	0.967	0.958	0.969



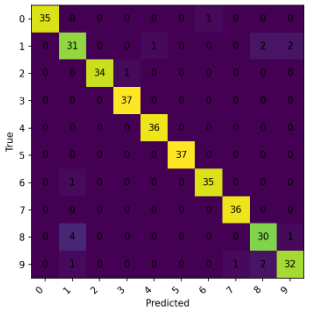
knn



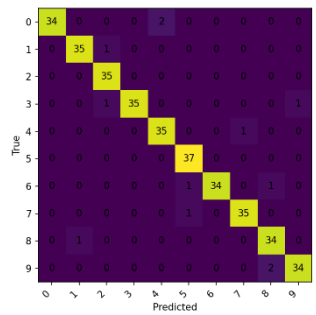
svm



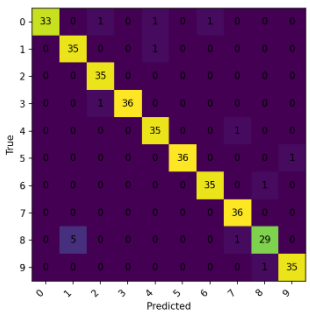
logreg



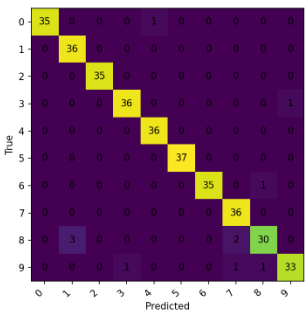
lda



qda



mlp



rf

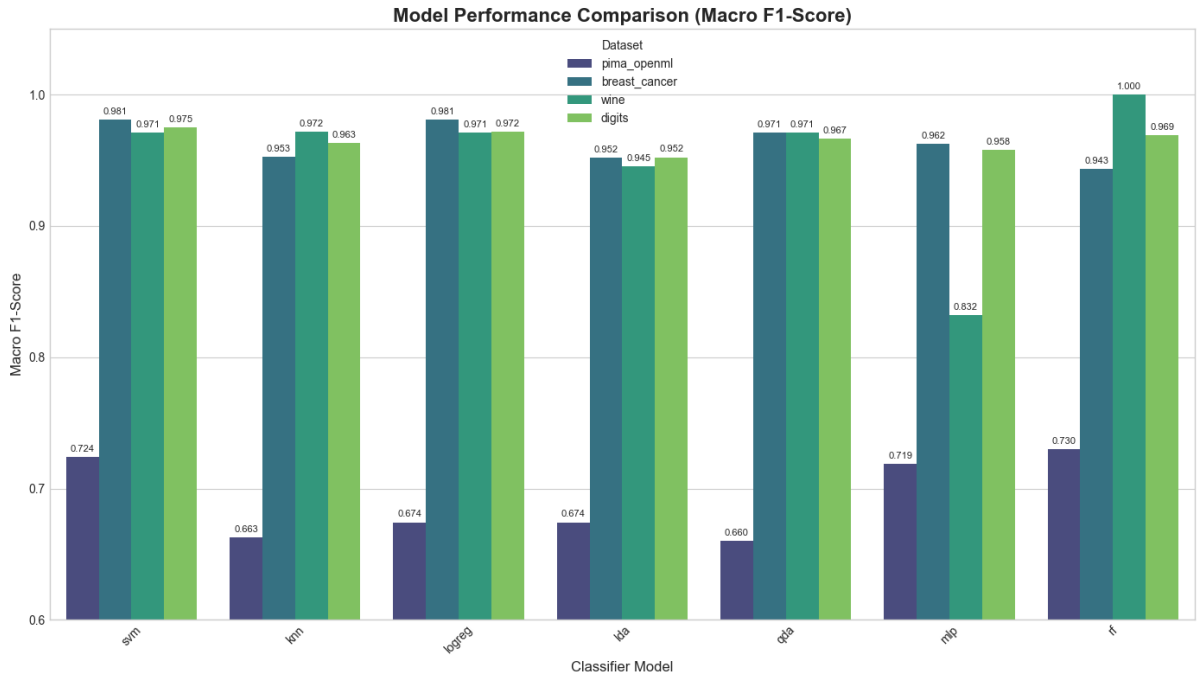
Confusion Matrices for Digits

5. Analysis

From the experimental results, I conducted a more in-depth analysis and have used visualizations to compare the performance of the models.

5.1. Overall Model Performance Across Datasets

To get a high-level view of how all the models performed, I created a visual comparison of their Macro F1-Scores across the four datasets.

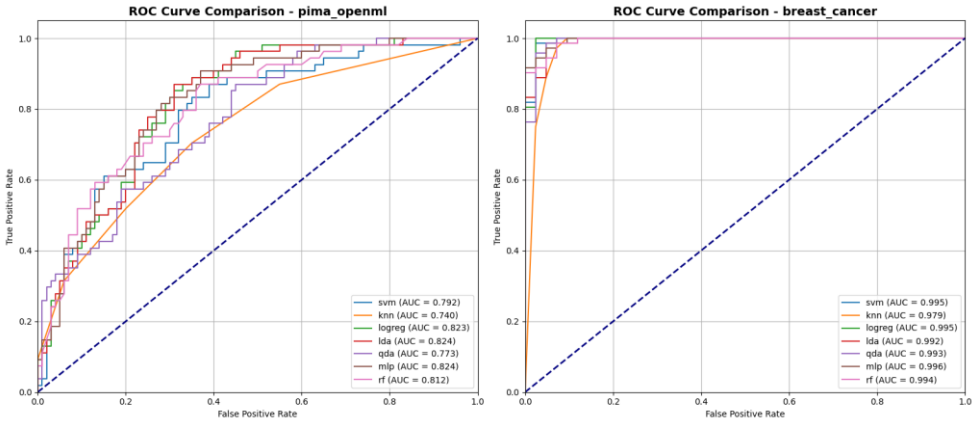


From the chart, a few key points become clear:

- Both SVM and Random Forest consistently delivered top-tier performance across all four datasets. They were particularly outstanding on datasets with relatively clear features, like breast_cancer, wine, and digits. This showcases their strong generalization capabilities and robustness in different scenarios.
- The MLP's performance was acceptable on pima_openml and digits, but its F1-Score on the wine dataset was significantly lower than the other models. This confirms my earlier hypothesis: as a complex model, MLP's performance is highly dependent on its hyperparameters, and without fine-tuning, its results can be unstable.
- k-NN had its weakest performance on the more challenging pima_openml dataset but was an upper-middle performer on the other three. This suggests that k-NN is more sensitive to the complexity and noise within the feature space.

5.2 A Deeper Look at Each Dataset's Results

Below are the comparative ROC curve plots for the pima_openml and breast_cancer datasets.



The pima_openml dataset proved to be the most challenging in this experiment, with no model surpassing 80% accuracy.

- AUC vs. Accuracy: On the test set, while LDA and MLP didn't have the highest accuracy, they achieved the highest AUC scores of 0.824. This indicates they had the strongest ability to distinguish between positive and negative classes. We can see from their ROC curves that the area is significantly larger than the baseline, confirming their strong discriminative power.

The features in the breast_cancer dataset are highly discriminative, which allowed most models to achieve accuracies above 95%.

- Near-Perfect Classification: SVM and Logistic Regression reached an accuracy of 98.2% on the test set with an impressive AUC of **0.995**. As seen in their confusion matrices, the vast majority of samples were classified correctly, with each model making only two errors.

On the two multi-class datasets, wine and digits, I observed similar trends.

- The Random Forest model achieved a perfect 100% accuracy on the wine dataset. Its confusion matrix shows a perfect diagonal, indicating zero misclassifications. This once again demonstrates the power of ensemble methods for this type of problem.
- SVM was the top performer on the digits dataset, with an accuracy of 97.5%. Looking at the confusion matrix, most errors were concentrated on visually similar digit pairs, such as misclassifying an '8' as a '1' or '3'.
- MLP's Struggles and Improvement: The untuned MLP performed the worst on the wine dataset, and its confusion matrix shows considerable confusion between classes 1 and 2. This contrasts with its relatively decent performance on the digits dataset, highlighting the importance of tuning a neural network's architecture and parameters for different tasks. On the wine dataset, I successfully improved its F1-score from 0.832 to 0.9353 through grid search.

```
Loading dataset: wine
Initializing model and parameter grid for: mlp
Starting GridSearchCV with 5-fold CV...
Fitting 5 folds for each of 9 candidates, totalling 45 fits
--- Grid Search Results ---
Best parameters found: {'mlpclassifier__alpha': 1e-05,
'mlpclassifier__hidden_layer_sizes': (64, 32)}
Best f1_macro score: 0.9353
```

```
param_grid = {
    'mlpclassifier__hidden_layer_sizes': [(64,), (128,), (64, 32)],
    'mlpclassifier__alpha': [1e-5, 1e-4, 1e-3],
}
```

5.1. code list

Main.py

```
import argparse, os, numpy as np
from src.datasets import load_dataset
from src.split import stratified_holdout
from src.evaluate import save_confmat, binary_roc_auc, summarize_metrics, dump_table
from src.classifiers.svm import SVM
from src.classifiers.knn import KNN
from src.classifiers.logistic import Logistic
from src.classifiers.lda_qda import LDA, QDA
from src.classifiers.mlp import MLP
from src.classifiers.rf import RF
from sklearn.model_selection import StratifiedKFold
import numpy as np

def get_models(names):
    out={}
    for n in names:
        if n=="svm": out[n]=SVM(kernel="rbf", C=1.0, gamma="scale",
probability=False)
        elif n=="knn": out[n]=KNN(n_neighbors=5, weights="uniform")
        elif n=="logreg": out[n]=Logistic(C=1.0)
        elif n=="lda": out[n]=LDA(solver="svd")
        elif n=="qda": out[n]=QDA(reg_param=0.01)
        elif n=="mlp": out[n]=MLP(hidden_layer_sizes=(128,), alpha=1e-4, max_iter=500)
        elif n=="rf": out[n]=RF(n_estimators=300)
        else: raise ValueError(f"unknown model {n}")
    return out

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--dataset", required=True, help="breast_cancer|wine|iris|digits or
path.csv:Target")
    ap.add_argument("--models", nargs="+", default=["svm","knn","logreg"])
    ap.add_argument("--outdir", default="reports")
    ap.add_argument("--cv", type=int, default=0, help="0=off, else k-fold")

    args = ap.parse_args()

    X, y, meta = load_dataset(args.dataset)
    Xtr, Xte, ytr, yte = stratified_holdout(X, y, test_size=0.2)
    if args.cv and args.cv > 1:
        skf = StratifiedKFold(n_splits=args.cv, shuffle=True, random_state=42)
        for name, mdl in get_models(args.models).items():
            accs, f1s = [], []
            for tr, va in skf.split(X, y):
                mdl.fit(X[tr], y[tr])
                pred = mdl.predict(X[va])
                accs.append((pred==y[va]).mean())
                from sklearn.metrics import f1_score
                f1s.append(f1_score(y[va], pred, average="macro"))
```

```
                row = {"dataset": meta["name"], "model": name,
                       "cv": args.cv,
                       "cv_acc_mean": float(np.mean(accs)), "cv_acc_std":
float(np.std(accs)),
                       "cv_macro_f1_mean": float(np.mean(f1s)), "cv_macro_f1_std":
float(np.std(f1s))}
                dump_table(row, os.path.join(args.outdir, "tables",
f"{meta['name']}_cv.csv"))
                print(row)

    models = get_models(args.models)
    for name, mdl in models.items():
        mdl.fit(Xtr, ytr)
        yhat = mdl.predict(Xte)
        disc = mdl.discriminant(Xte)

        fig_dir = os.path.join(args.outdir, "figures")
        save_confmat(yte, yhat, classes=[str(i) for i in sorted(set(y))],
                     out_png=os.path.join(fig_dir,
f"{meta['name']}_{name}_confmat.png"))
        aucv=None
        if meta["is_binary"] and disc.ndim==1:
            aucv = binary_roc_auc(yte, disc, out_png=os.path.join(fig_dir,
f"{meta['name']}_{name}_roc.png"))

        row = {"dataset": meta["name"], "model": name, **summarize_metrics(yte, yhat,
disc if disc.ndim==1 else None, meta["is_binary"])}
        dump_table(row, os.path.join(args.outdir, "tables", f"{meta['name']}.csv"))
        print(row)

if __name__=="__main__":
    main()
```

datasets.py

```
import numpy as np, pandas as pd
from sklearn.datasets import load_breast_cancer, load_wine, load_iris, load_digits
from sklearn.datasets import fetch_openml
def _from_sklearn(loader, name):
    d = loader()
    X = d["data"].astype(float)
    y = d["target"].astype(int)
    meta = dict(name=name, n_classes=len(np.unique(y)),
                is_binary=len(np.unique(y))==2,
                feature_names=getattr(d, "feature_names", [f"f{i}" for i in
range(X.shape[1])]))
    return X, y, meta
def _from_openml_pima():
    df = fetch_openml(name="diabetes", version=1, as_frame=True).frame
    y = (df['class'] == 'tested_positive').astype(int).to_numpy()
    X = df.drop(columns=['class']).astype(float).to_numpy()
    meta = dict(name="pima_openml", n_classes=2, is_binary=True,
feature_names=list(df.columns.drop('class')))
    return X, y, meta
def _from_csv(path, target_col):
    df = pd.read_csv(path)
    if target_col not in df.columns:
        raise ValueError(f"Target column '{target_col}' not in CSV columns:
{list(df.columns)}")
    y = df[target_col].astype(int).to_numpy()
    X = df.drop(columns=[target_col]).astype(float).to_numpy()
    meta = dict(name=path, n_classes=len(np.unique(y)),
                is_binary=len(np.unique(y))==2,
                feature_names=[c for c in df.columns if c != target_col])
    return X, y, meta
def load_dataset(name: str):
    if ":" in name:
        path, target = name.split(":", 1)
        if path.lower().endswith(".csv"):
            return _from_csv(path, target)
    key = name.lower()
    if key in ["breast_cancer", "cancer", "bc"]:
        return _from_sklearn(load_breast_cancer, "breast_cancer")
    if key == "wine":
        return _from_sklearn(load_wine, "wine")
    if key == "iris":
        return _from_sklearn(load_iris, "iris")
    if key == "digits":
        return _from_sklearn(load_digits, "digits")
    if key in ["pima", "pima_openml", "diabetes_pima"]:
        return _from_openml_pima()

    raise ValueError(f"Unknown dataset: {name}")
```

split.py

```
import numpy as np
```

```
from sklearn.model_selection import StratifiedKFold, train_test_split

RANDOM_STATE = 42

def stratified_holdout(X, y, test_size=0.2, rs=RANDOM_STATE):
    return train_test_split(X, y, test_size=test_size, stratify=y, random_state=rs)

def stratified_kfold(n_splits=5, rs=RANDOM_STATE):
    return StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=rs)
```

evaluate.py

```
# src/evaluate.py
import matplotlib
matplotlib.use("Agg")
import os, numpy as np, matplotlib.pyplot as plt, pandas as pd
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc,
accuracy_score, f1_score

def save_confmat(y_true, y_pred, classes, out_png):
    cm = confusion_matrix(y_true, y_pred)
    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation="nearest")
    ax.set_xlabel("Predicted"); ax.set_ylabel("True")
    ax.set_xticks(range(len(classes))); ax.set_xticklabels(classes, rotation=45, ha="right")
    ax.set_yticks(range(len(classes))); ax.set_yticklabels(classes)
    for i in range(len(classes)):
        for j in range(len(classes)):
            ax.text(j, i, cm[i,j], ha="center", va="center")
    fig.tight_layout(); os.makedirs(os.path.dirname(out_png), exist_ok=True)
    fig.savefig(out_png, dpi=150); plt.close(fig)

def binary_roc_auc(y_true, scores, out_png):
    fpr, tpr, _ = roc_curve(y_true, scores)
    A = auc(fpr, tpr)
    fig, ax = plt.subplots()
    ax.plot(fpr, tpr, label=f"AUC={A:.3f}")
    ax.plot([0,1],[0,1], linestyle="--")
    ax.set_xlabel("FPR"); ax.set_ylabel("TPR"); ax.legend()
    fig.tight_layout(); os.makedirs(os.path.dirname(out_png), exist_ok=True)
    fig.savefig(out_png, dpi=150); plt.close(fig)
    return A

def summarize_metrics(y_true, y_pred, scores=None, is_binary=False):
    acc = accuracy_score(y_true, y_pred)
    f1m = f1_score(y_true, y_pred, average="macro")
    out = {"accuracy": acc, "macro_f1": f1m}
    if is_binary and scores is not None and np.ndim(scores)==1:
        fpr, tpr, _ = roc_curve(y_true, scores); out["auc"] = auc(fpr, tpr)
    return out

def dump_table(row_dict, csv_path):
    os.makedirs(os.path.dirname(csv_path), exist_ok=True)
    df = pd.DataFrame([row_dict])
    if os.path.exists(csv_path):
        base = pd.read_csv(csv_path)
        df = pd.concat([base, df], ignore_index=True)
    df.to_csv(csv_path, index=False)
```

Classifiers

base.py

```
class ClassifierBase:
    def fit(self, X_train, y_train): raise NotImplementedError
    def predict(self, X_test): raise NotImplementedError
    def discriminant(self, X_test): raise NotImplementedError
```

knn.py

```
# src/classifiers/knn.py
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from .base import ClassifierBase
import numpy as np

class KNN(ClassifierBase):
    def __init__(self, n_neighbors=5, weights="uniform"):
        self.model = make_pipeline(StandardScaler(),
                                   KNeighborsClassifier(n_neighbors=n_neighbors,
weights=weights))
    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        proba = self.model.predict_proba(X)
        return proba[:,1] if proba.shape[1]==2 else proba
```

svm.py

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from .base import ClassifierBase

class SVM(ClassifierBase):
    def __init__(self, kernel="rbf", C=1.0, gamma="scale", probability=False,
random_state=42):
        self.model = make_pipeline(StandardScaler(),
                                   SVC(kernel=kernel, C=C, gamma=gamma,
                                       probability=probability,
random_state=random_state))
    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        svc = self.model.named_steps["svc"]
        return svc.decision_function(self.model[:-1].transform(X)) if
hasattr(svc,"decision_function") else self.model.predict_proba(X)[:,1]
```

logistic.py

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from .base import ClassifierBase

class Logistic(ClassifierBase):
    def __init__(self, C=1.0, max_iter=1000, solver="lbfgs", class_weight=None):
        self.model = make_pipeline(
            StandardScaler(),
            LogisticRegression(C=C, max_iter=max_iter, solver=solver,
class_weight=class_weight)
        )
    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        proba = self.model.predict_proba(X)
        return proba[:,1] if proba.shape[1]==2 else proba
```

lda_qda.py

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,
QuadraticDiscriminantAnalysis
from .base import ClassifierBase
import numpy as np
class LDA(ClassifierBase):
    def __init__(self, solver="svd"):
        self.model = make_pipeline(StandardScaler(with_mean=True, with_std=True),
                                   LinearDiscriminantAnalysis(solver=solver))
    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        est = self.model.named_steps.get("lineardiscriminantanalysis")
        try: return est.decision_function(self.model[:-1].transform(X))
        except:
            proba = self.model.predict_proba(X)
            return proba[:,1] if proba.shape[1]==2 else proba
class QDA(ClassifierBase):
    def __init__(self, reg_param=0.01):
        self.model = QuadraticDiscriminantAnalysis(reg_param=reg_param)
    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        try: return self.model.decision_function(X)
        except:
            proba = self.model.predict_proba(X)
            return proba[:,1] if proba.shape[1]==2 else proba
```

mlp.py

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.neural_network import MLPClassifier
from .base import ClassifierBase

class MLP(ClassifierBase):
    def __init__(self, hidden_layer_sizes=(128,), alpha=1e-3, max_iter=1000,
random_state=42):
        self.model = make_pipeline(StandardScaler(),
                                   MLPClassifier(hidden_layer_sizes=hidden_layer_s
izes,
                                                 alpha=alpha, max_iter=max_iter,
                                                 early_stopping=True,
                                                 n_iter_no_change=20,
                                                 random_state=random_state))

    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        proba = self.model.predict_proba(X)
        return proba[:,1] if proba.shape[1]==2 else proba
```

rf.py

```
from sklearn.ensemble import RandomForestClassifier
from .base import ClassifierBase

class RF(ClassifierBase):
    def __init__(self, n_estimators=300, max_depth=None, random_state=42):
        self.model = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth,
                                            random_state=random_state, n_jobs=-1)
    def fit(self, X, y): self.model.fit(X, y); return self
    def predict(self, X): return self.model.predict(X)
    def discriminant(self, X):
        proba = self.model.predict_proba(X)
        return proba[:,1] if proba.shape[1]==2 else proba
```