IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UETAF IML

Practical on SVMs

Yannis Haralambous (IMT Atlantique)

October 20, 2021

In this practical we will first scrutinize an elementary example of SVM and then see some SVM applications under Python. The packages used are `scikit-learn` (and its subpackages `svm` et `datasets`), `numpy` and `matplotlib`.

1 An elementary SVM

In the Euclidean plane $\mathbb{R}^2$ take individuals $(1,1)$, $(1,5)$, $(5,1)$ and $(5,5)$, assigned to classes $-1, 1, 1, 1$.

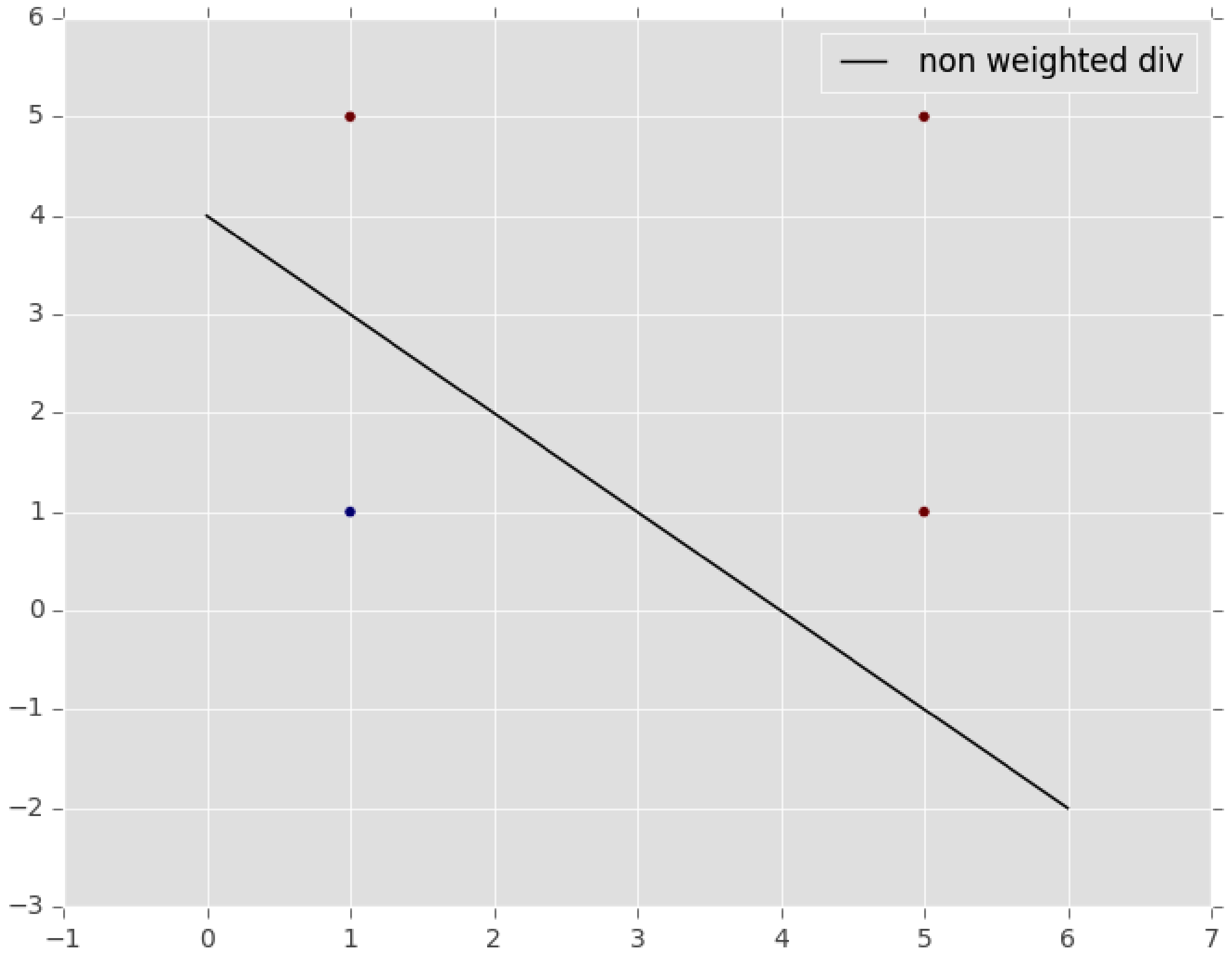
1) Manually calculate the vector (a, b) and the value of c of the maximal margin classifier straight line.

2) Implement:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.style.use("ggplot")
from sklearn import svm
X = np.array([[1, 1], [1, 5], [5, 1], [5, 5]])
y = [-1,1,1,1]
clf = svm.SVC(kernel='linear')
clf.fit(X, y)
print("clf.support_vectors_ = ")
print(clf.support_vectors_)
print(f"{clf.n_support_ = !s}")
print(f"{              w = !s}")
w = clf.coef_[0]
print(w)
a = -w[0] / w[1]
xx = np.linspace(0,6)
yy = a * xx - clf.intercept_[0] / w[1]

fig, ax = plt.subplots()
ax.plot(xx, yy, "k-", label="non weighted div")
ax.scatter(X[:, 0], X[:, 1], c=y)
ax.legend()
```

to obtain



What is the meaning of the obtained results:

```
clf.support_vectors_ =
[[1. 1.]
 [1. 5.]
 [5. 1.]]
clf.n_support_ = [1 2]
            w = [0.49975586 0.49975586]
```

SOLUTION La droite est antidiagonale donc le vecteur normal est diagonal (a, a). La distance géométrique entre les vecteurs supports et la droite est de $\sqrt{2}$ (Pythagore). On veut que $ax + by + c$ soit 1 aux points $(5, 1)$ et $(1, 5)$. La formule du cours est $d(M, D) = |ax + by + c|/||(a, b)||$, donc pour ces deux points, $||\alpha|| = 1/d(M, D) = 1/\sqrt{2}$. Donc le vecteur normal est $(0.5, 0.5)$. Reste à trouver c : $0.5 \cdot 5 + 0.5 \cdot 1 + c = 1 \Rightarrow c = -2$.

2 Iris

We will start with the famous data set “Iris”: the size in centimeters of petals and other parts of some flowers. For every individual we have four number and the class, among*Iris setosa*, *Iris versicolor* and *Iris virginica*. We have 150 individuals and equidistributed classes.

Do

```
from sklearn import svm
from sklearn import datasets
from sklearn.model_selection import train_test_split
# Load data
iris = datasets.load_iris()

# Extract arrays
X, y = iris.data, iris.target

# Extract some useful information
num_classes = len(iris.target_names)
classes_labels = sorted(set(iris.target))

# Initialize classfier
clf = svm.SVC()
```

Out of the `iris` data create a training set `X_train,y_train` with 100 random individuals and a test set `X_test,y_test` with what remains. *Advice: use the `shuffle` function from package `random`.*

SOLUTION

```
train_size = 100
X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=train_size, shuffle=True
)
```

Launch the SVM by writing

```
clf.fit(X_train, y_train)
```

For predictions, use

```
y_pred = clf.predict(X_test)
```

Calculate, using Python list comprehension the precision and recall of each class.

SOLUTION

```
from sklearn.metrics import precision_score, recall_score


def precision_recall_multilabels(y_true, y_pred, labels):
    recalls = []
    precisions = []
    for label in labels:

        pos_true = y_true == label
        pos_pred = y_pred == label

        # By hand
        true_pos = pos_pred & pos_true
        recalls.append(np.sum(true_pos) / np.sum(pos_true))
        precisions.append(np.sum(true_pos) / np.sum(pos_pred))

        # With sklearn
        # precisions.append(precision_score(pos_test, pos_pred))
        # recalls.append(recall_score(pos_test, pos_pred))

    return precisions, recalls
```

```
precisions, recalls = precision_recall_multilabels(y_test, y_pred, classes_labels)

print(f"{precisions = }")
print(f"{   recalls = }")
```

Check result against `sklearn.metrics.classification_report`.

SOLUTION

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred, digits=4))
```

By using class `KFold` from package `sklearn.cross_validation`, write the code a 10-crossed validation and obtain average precision and recall for each class.

Use various kernel types and parameter values to see how the performances vary.

You have the choice between four kernels (option `kernel` of `SVC`) :

1. linear `linear` $k(x, x') = \langle x, x' \rangle$;
2. polynomial `pol` $k(x, x') = (\gamma \cdot \langle x, x' \rangle + r)^d$;
3. radial `rbf` $k(x, x') = e^{-\gamma \|x - x'\|}$ (par défaut) ;
4. sigmoid `sigmoid` $k(x, x') = \mathrm{th}(\gamma \cdot \langle x, x' \rangle)$.

Parameters γ, d and r are written `gamma`, `degree` and `coef0`, resp. The cost parameter C (see in the class) is written `cost`.

SOLUTION

```
from sklearn.model_selection import KFold
# Once you have filled the `clfs_results` dictionary below
# call `kfold_multimodels_report(clfs_results)`


def kfold_multimodels_report(clfs_results: dict[str, dict[str, list[list[float]]]]):
    """
    Prints a report for the results of experiments on multiple models,
    each one evaluated using k-fold cross-validation.

    The results of the experiments should be given as the 'clfs_stats'
    argument, with the following structure:

    {
        "clf_name1": {"metric1": list[list], "metric2": list[list], ...},
        "clf_name2": {"metric1": list[list], "metric2": list[list], ...},
        ...
    }

    with each list[list] being of shape (num_folds, num_classes).
    """
    clfs_stats = kfold_summarize_results(clfs_results)
    with np.printoptions(precision=2, floatmode="fixed"):
        for clf_name, clf_stats in clfs_stats.items():
            print(f"{clf_name:<15}")
            for metric_name, stats in clf_stats.items():
                print(f"{metric_name:>15}")
                for stat_name, data in stats.items():
                    print(f"{stat_name:>20}: {data}")


def kfold_summarize_results(clfs_results):
    """Computes stats on results of multi-models k-folds experiments.

    Takes:

    {
        "clf_name1": {"metric1": list[list], "metric2": list[list], ...},
        "clf_name2": {"metric1": list[list], "metric2": list[list], ...},
        ...
    }

    Returns:

    {
        "clf_name1": {"metric1": {"mean": value, "std": value ...}, ...},
        ...
    }
    """
    clfs_stats = {clf_name: {} for clf_name in clfs_results}
    for clf_name, clf_results in clfs_results.items():
        for metric, data in clf_results.items():
            clfs_stats[clf_name][metric] = {
                "mean": np.mean(data, axis=0),
                "std": np.std(data, axis=0),
            }
    return clfs_stats
def kfold_precisions_recalls(X, y, labels, clf, kf: KFold):
    """Returns the history of precisions and recalls through K-fold training

    Parameters
    ----------
    X, y : data
    labels : list[int]
    clf : classifier
    kf : KFold instance

    Returns
    -------
    precisions : list[list], shape (num_folds, len(labels))
    recalls : list[list], shape (num_folds, len(labels))
    """
    precisions, recalls = [], []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)

        precisions_, recalls_ = precision_recall_multilabels(y_test, y_pred, labels)

        precisions.append(precisions_)
        recalls.append(recalls_)

    return precisions, recalls
X, y = iris.data, iris.target
num_classes = len(iris.target_names)
classes_labels = sorted(set(iris.target))
```

```
clfs = {
    "linear": svm.SVC(kernel="linear", C=1.0),
    "poly2": svm.SVC(kernel="poly", C=1.0, degree=2, gamma="scale", coef0=0.0),
    "poly3": svm.SVC(kernel="poly", C=1.0, degree=3, gamma="scale", coef0=0.0),
    "poly4": svm.SVC(kernel="poly", C=1.0, degree=4, gamma="scale", coef0=0.0),
    "rbf": svm.SVC(kernel="rbf", C=1.0, gamma="scale"),
    # "sigmoid": svm.SVC(kernel="sigmoid", C=1.0, gamma="scale", coef0=0.0),
}

clfs_results = {clf_name: {"precisions": None, "recalls": None} for clf_name in clfs}
kf = KFold(n_splits=10, shuffle=True, random_state=34)

for clf_name, clf in clfs.items():
    precisions, recalls = kfold_precisions_recalls(X, y, classes_labels, clf, kf)

    clfs_results[clf_name]["precisions"] = precisions
    clfs_results[clf_name]["recalls"] = recalls

kfold_multimodels_report(clfs_results)
```

3 Chronic Kidney Disease

3.1 Data preparation

- Fetch the file `chronic_kidney_disease_full.arff` from Moodle. Read the introductory comments and get acquainted with the ARFF data format.

  (For more information see `http://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease`)
- Read the data into a `pandas.DataFrame` named `ckd`
- Clean up the data
- Decode strings: `pandas.DataFrame.columns`, `pandas.DataFrame.dtypes`
- Handle nodata values (replace them with actual `np.nan`): `pandas.DataFrame.replace`.
- Convert data types that are still wrong: `pandas.DataFrame.apply`, `pandas.to_numeric`.
- Convert categorical features to corresponding data type: `pandas.DataFrame.astype`.

SOLUTION

```
from scipy.io import arff
import pandas as pd
ckd_path = "chronic_kidney_disease_full.arff"
nodataval = "?"

# Read
data, meta = arff.loadarff(ckd_path)
ckd = pd.DataFrame(data)
num_samples, num_features = ckd.shape

# Decode strings
is_str_cols = ckd.dtypes == object
str_columns = ckd.columns[is_str_cols]
ckd[str_columns] = ckd[str_columns].apply(lambda s: s.str.decode("utf-8"))
```

```
# Handle nodata values
ckd = ckd.replace(nodataval, np.nan)

# Convert remaining false string columns
other_numeric_columns = ["sg", "al", "su"]
ckd[other_numeric_columns] = ckd[other_numeric_columns].apply(pd.to_numeric)

# Use categorical data type
categoric_columns = pd.Index(set(str_columns) - set(other_numeric_columns))
ckd[categoric_columns] = ckd[categoric_columns].astype("category")

ckd
```

- Remove the “ground-truth” column and store its values aside: `pandas.DataFrame.drop`

SOLUTION

```
y = ckd["class"]
ckd = ckd.drop(columns="class")
```

- For missing numeric values, replace with the feature average: `pandas.DataFrame.fillna`
- For missing categorical values, replace with the feature most occuring value `pandas.DataFrame.mode` (hint: `ckd.mode().iloc[0]`)

SOLUTION

```
# Check the number of missing values
ckd.isna().sum(axis=0)

fillna_mean_cols = pd.Index(
    set(ckd.columns[ckd.dtypes == "float64"]) - set(other_numeric_columns)
)
fillna_most_cols = pd.Index(
    set(ckd.columns[ckd.dtypes == "category"]) | set(other_numeric_columns)
)
assert set(fillna_mean_cols.union(fillna_most_cols)) == set(ckd.columns)

ckd[fillna_mean_cols] = ckd[fillna_mean_cols].fillna(ckd[fillna_mean_cols].mean())
ckd[fillna_most_cols] = ckd[fillna_most_cols].fillna(
    ckd[fillna_most_cols].mode().iloc[0]
)
ckd
```

- One-hot encode categorical features: `pandas.get_dummies`
- Normalize the data

SOLUTION

```
ckd = (ckd - ckd.mean()) / (ckd.std())
```

- Get acquainted with the input format of SVM Light, convert the data into that format, save them in a file `data.dat`.
- Convert the "ground-truth" data into the right numeric format into a numpy array `y`: `pandas.Series.map`, lambda expressions, `pandas.Series.to_numpy`
- Extract the data in a numpy array `X` `pandas.DataFrame.to_numpy`
- Write data.dat from `X` and `y` `str.join`, f-strings `open`, `io.TextIOBase.write`

SOLUTION

```
trans_table = {"notckd": -1, "ckd": 1}
y = y.map(lambda x: trans_table[x]).to_numpy()
X = ckd.to_numpy()
features_names = list(ckd.columns)

def data_to_dat_str(X, y, features_names=None):
    def row_to_dat_str(row, label):
        values = " ".join(f"{ft_idx:d}:{val:.12f}" for ft_idx, val in enumerate(row, 1))
        return f"{label} {values}"

    header = "" if features_names is None else "#" + " ".join(features_names) + "\n"
    body = "\n".join(row_to_dat_str(row, label) for row, label in zip(X, y))
    return header + body


def write_dat_file(fpath, X, y, features_names=None):
    with open(fpath, "w") as f:
        f.write(data_to_dat_str(X, y, features_names))

from pathlib import Path

data_root = Path("data")
data_root.mkdir(exist_ok=True)

dat_fpath = data_root / "data.dat"
write_dat_file(dat_fpath, X, y, features_names)

!head -n 3 $dat_fpath

!tail -n 2 $dat_fpath
```

3.2 Data preparation for cross validation

Read `data.dat` and shuffle lines. Divide in ten parts, put every test corpus into `test`i`.dat` for $i = 0, \ldots, 9$, and every training corpus into `train`i`.dat` for $i = 0, \ldots, 9$.

SOLUTION

```
import random, os

n_splits = 10
fpaths = [
    (data_root / f"train_{fold_idx:02d}.dat", data_root / f"test_{fold_idx:02d}.dat")
    for fold_idx in range(n_splits)
]
kf = KFold(n_splits=n_splits, shuffle=True, random_state=34)

for (train_path, test_path), (train_index, test_index) in zip(fpaths, kf.split(X)):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    write_dat_file(train_path, X_train, y_train, features_names)
    write_dat_file(test_path, X_test, y_test, features_names)
```

3.3 Cross-validation and results

Download and install SVM Light from `https://www.cs.cornell.edu/people/tj/svm_light/index.html` Find out the command line syntax and run SVM Light ten times from within Python on the `test_xx.dat` and `train_xx.dat` files. Capture the data returned from SVM Light, and parse it to extract runtime, error, precision and recall `subprocess.run` `subprocess.CompletedProcess.stdout`, `bytes.decode` `re.match`.

Calculate the average runtime, error, precision and recall. Use various kernels and play with the parameters to see whether the results can be improved, without affecting performance too much.

SOLUTION

```
import subprocess, os, re, shlex


def run_command(cmd: str, args: str) -> subprocess.CompletedProcess:
    cmd_line_parsed = shlex.split(f"{cmd} {args}")
    return subprocess.run(cmd_line_parsed, capture_output=True)


def parse_svm_light_stdout(stdout: str) -> dict[str, str]:
    """See https://regex101.com/r/fnnhtb/1 for help"""

    number = r"\d+(?:\.\d+)?"
    regexes = {
        "runtime": f"Runtime in cpu-seconds: ({number})",
        "error": f"XiAlpha-estimate of the error: error<=({number})",
        "precision": f"XiAlpha-estimate of the recall: recall=>({number})",
        "recall": f"XiAlpha-estimate of the precision: precision=>({number})",
    }

    results = {}
    for line in stdout.split("\n"):
        for name, regex in regexes.items():
            if (m := re.match(regex, line)) is not None:
                results[name] = m.group(1)
                break
    return results

svm_exe = (Path(".") / "svm_light" / "svm_learn").absolute()
model_path = data_root / "model"

args_fmt = f"-z c -c 1.0 -t 1 -d 2 {{}} {model_path}"

results = []
for train_path, test_path in fpaths:

    args = args_fmt.format(train_path)
    p = run_command(svm_exe, args)
    stdout = p.stdout.decode()
    res = parse_svm_light_stdout(stdout)

    results.append(res)

results = pd.DataFrame.from_records(results).astype(float)
results

results.describe()
```

Compare with scikit-learn

SOLUTION

```
from sklearn.metrics import precision_score, recall_score

clf = svm.SVC(kernel="poly", degree=2)

precisions = []
recalls = []
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    recalls.append(recall_score(y_test, y_pred))
    precisions.append(precision_score(y_test, y_pred))

sklearn_results = pd.DataFrame({"precision": precisions, "recall": recalls})
sklearn_results
```

4 SPAM

For those who finished the previous exercise and are still motivated for another example, which they can continue at home, fetch the SPAM dataset from Hewlett-Packard:

`https://archive.ics.uci.edu/ml/datasets/Spambase`

It contains 4 061 individuals with 57 features and the class (SPAM or not SPAM). There are 39,4% SPAM individuals and 60,59% not SPAM individuals. It is preferable to avoid false positives, see if you can get zero false positives and check how many SPAMs pass the filter then.