# Machine Learning

Written exam • Solutions and grading scheme

Duration: 1 hour • 20 points

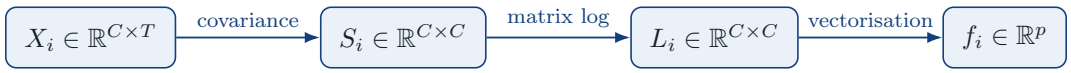
## INSTRUCTIONS

- No documents, no calculator, no electronic device. Answer directly on this sheet.
- Each multiple choice question has **exactly one** correct answer among four. Tick one box. A wrong answer or no answer gives 0 point. There are no negative points.
- Code is written in Python with NumPy. A NumPy reminder is given on the last page. Minor syntax errors are not penalised: the logic is graded.
- The points of each question are given next to its number.

## CONTEXT

We observe $m$ recordings. Each recording is a matrix $X_i \in \mathbb{R}^{C\times T}$: $C$ sensors measure a signal at $T$ time instants. Each recording comes with a class label $y_i \in \{0, 1, \dots, K-1\}$ and a continuous target $z_i \in \mathbb{R}$.

Instead of using the raw signals, we describe each recording by how its sensors vary together, i.e. by its covariance matrix. This approach is standard, for instance, to decode brain signals (EEG) in brain-computer interfaces. The pipeline has three steps:




The vectors $f_i$ are stacked as the rows of a feature matrix $F \in \mathbb{R}^{m\times p}$. We then use $F$ to predict $z$ (regression) and $y$ (classification).

## PART 1. FROM SIGNALS TO FEATURES — 6 pts

### QUESTION 1 — 1.5 pts

Complete the function `covariance`. Each row of $X$ (one sensor) is first centred: we subtract its mean over time, which gives $\bar{X}$. The covariance matrix is then

$$S = \frac{1}{T-1}\,\bar{X}\bar{X}^\top.$$

```python
def covariance(X):
    """X: array of shape (C, T). Returns S, array of shape (C, C)."""
```

```python
    T = X.shape[1]
    Xc = X - X.mean(axis=1, keepdims=True)   # centre each sensor
    return Xc @ Xc.T / (T - 1)
```

**Solution.** 0.5 for the centring along the time axis, 0.5 for the product $\bar{X}\bar{X}^\top$, 0.5 for the normalisation by $T-1$.

**QUESTION 2** 1.5 pts

A covariance matrix is symmetric with positive eigenvalues $\lambda_1, \dots, \lambda_C$. Denoting by $U$ the matrix of its eigenvectors, it can be written $S = UDU^\top$ with $D = \mathrm{diag}(\lambda_1, \dots, \lambda_C)$. Its *matrix logarithm* is obtained by taking the logarithm of each eigenvalue:

$$L = \log S = U \,\mathrm{diag}(\log \lambda_1, \dots, \log \lambda_C)\, U^\top.$$

Complete the function `logm_spd`.

```
def logm_spd(S):
    """S: covariance matrix of shape (C, C). Returns log(S), of shape (C, C)."""
```

```
    lam, U = np.linalg.eigh(S)
    return U @ np.diag(np.log(lam)) @ U.T
```

**Solution.** 0.5 for `eigh` with the right order of outputs, 0.5 for log applied to the eigenvalues only, 0.5 for the product $U\,\mathrm{diag}(\cdot)\,U^\top$.

**QUESTION 3** 1 pt

The matrix $L$ is symmetric, so its lower triangle repeats its upper triangle. Complete the function `vectorize`, which returns the entries of the upper triangle of $L$, diagonal included, as a vector.

```
def vectorize(L):
    """L: symmetric matrix of shape (C, C). Returns a vector of size C(C+1)/2."""
```

```
    i, j = np.triu_indices(L.shape[0])
    return L[i, j]
```

The feature matrix is then built as follows:

```
F = np.array([vectorize(logm_spd(covariance(X))) for X in recordings])
```

**QUESTION 4** 1 pt

Some recordings are very short, with fewer time instants than sensors ($T < C$). What happens in `logm_spd`?

- ☐ **A** $S$ is no longer symmetric, so `eigh` raises an error.
- ☐ **B** $S$ has negative eigenvalues, so the logarithm returns `nan`.
- ☑ **C** $S$ is not invertible: some eigenvalues are zero, so the logarithm returns $-\infty$.
- ☐ **D** Nothing special: the features are simply less precise.

**Solution.** The centred matrix $\bar{X}$ has at most $T - 1 < C$ independent columns, so $S$ has rank at most $T - 1$. At least one eigenvalue is zero (in practice, a tiny or slightly negative number due to rounding).

**QUESTION 5** 1 pt

Write the line to add at the beginning of `logm_spd` to fix this problem, with a small constant `eps`.

```
    S = S + eps * np.eye(S.shape[0])   # all eigenvalues become >= eps
```

## PART 2. GOOD PRACTICES 2.5 pts

**QUESTION 6** 1 pt

We want to standardise the features before training a model. Which procedure is correct?

- ☑ **A** Split into train and test sets, compute the mean and standard deviation on the train set, then use them to standardise both sets.
- ☐ **B** Compute the mean and standard deviation on the whole data set, standardise, then split into train and test sets.
- ☐ **C** Split, then standardise each set with its own mean and standard deviation.
- ☐ **D** Standardise only the test set, since the model must not see it.

**Solution.** The model must see the test points through the same transformation as the train points. With its own statistics, the test set is rescaled differently: a feature value $v$ is mapped to different numbers in the two sets, so the decision rule learned on the train set no longer applies to the same quantities. It also cannot be done in practice for a single new point, which has no standard deviation.

**QUESTION 7** 1.5 pts

A student evaluates an SVM with the code below. Explain the problem with this code, and how to fix it.

```
scaler = StandardScaler()
F_std = scaler.fit_transform(F)
F_train, F_test, y_train, y_test = train_test_split(F_std, y, test_size=0.25)
model = SVC().fit(F_train, y_train)
print(model.score(F_test, y_test))
```

**Solution.** The scaler is fitted on the whole data set, test set included. Information from the test set leaks into the training, and the reported score is optimistic. (1)
Fix: split first, fit the scaler on the train set only, then apply this same scaler to the train and test sets. (0.5)

## PART 3. LINEAR MODELS: RIDGE AND LOGISTIC REGRESSION 4.5 pts

### Regression of the target $z$

We predict $z$ with a Ridge model $\hat{z} = Fw + b$. Its coefficients minimise the squared error plus the penalty $\lambda\|w\|^2$, where $\lambda > 0$ is a hyperparameter.

**QUESTION 8** 1 pt

We trained the model for several values of $\lambda$. The table gives the mean squared errors on the train set and on a validation set.

| $\lambda$ | $10^{-3}$ | $10^{-2}$ | $10^{-1}$ | 1 | 10 | 100 |
|---|---|---|---|---|---|---|
| Train error | 0.8 | 1.1 | 1.6 | 2.4 | 4.0 | 7.5 |
| Validation error | 9.5 | 6.2 | 3.9 | 3.1 | 4.4 | 7.9 |

Which value of $\lambda$ do you choose? For which values does the model overfit, and for which does it underfit?

**Solution.** $\lambda = 1$, which minimises the validation error. (0.5)
For $\lambda \leq 10^{-2}$ the model overfits (small train error, large validation error). For $\lambda = 100$ it underfits (both errors are large). (0.5)

### Evaluating a classifier: confusion matrix and decision threshold

To evaluate a classifier on $K$ classes, we use its *confusion matrix* $M \in \mathbb{N}^{K \times K}$: the entry $M_{kl}$ counts the test recordings of true class $k$ predicted in class $l$. A perfect classifier has a diagonal confusion matrix.

**QUESTION 9** 2 pts

Complete the function `confusion_matrix`.

```
def confusion_matrix(y_true, y_pred, K):
    """y_true, y_pred: arrays of shape (n,) in {0,...,K-1}. Returns M of shape (K, K)."""
```

```
    M = np.zeros((K, K), dtype=int)
    for k, l in zip(y_true, y_pred):
        M[k, l] += 1        # true class k, predicted class l
    return M
```

**Solution.** 0.5 for the initialisation, 0.75 for the loop over the pairs, 0.75 for incrementing the entry with the true class as row index.

We now consider only two classes, 0 and 1. A logistic regression model gives each recording a probability $p_i = P(y_i = 1 \mid f_i)$, and predicts class 1 when $p_i \geq t$, where the threshold is $t = 0.5$ by default. In the confusion matrix, $M_{01}$ counts the *false positives* (class 0 predicted as 1) and $M_{10}$ the *false negatives* (class 1 predicted as 0).

**QUESTION 10** 1.5 pts

We keep the same trained model and raise the threshold from $t = 0.5$ to $t = 0.8$. What happens on the test set?

- ☐ **A** Both $M_{01}$ and $M_{10}$ decrease or stay equal.
- ☐ **B** Both $M_{01}$ and $M_{10}$ increase or stay equal.
- ☑ **C** $M_{01}$ decreases or stays equal, and $M_{10}$ increases or stays equal.
- ☐ **D** Nothing can be said without training the model again with the new threshold.

**Solution.** The probabilities do not change, only the rule that turns them into classes. With $t = 0.8$, fewer recordings are predicted in class 1: some false positives become true negatives, and some true positives become false negatives. The threshold sets the trade-off between the two types of error, for instance when missing a class 1 is more costly.

## PART 4. NEAREST NEIGHBOURS AND SVM — 4 pts

**QUESTION 11** 2 pts

Complete the function `knn_predict`, which predicts the label of a new feature vector $f$ by a majority vote among its $k$ nearest training points (Euclidean distance). In case of a tie, any of the tied classes is accepted.

```
def knn_predict(F_train, y_train, f, k):
    """F_train: (m, p), y_train: (m,) in {0,...,K-1}, f: (p,). Returns a class."""
```

```
    d = np.sum((F_train - f) ** 2, axis=1)   # squared distances to all points
    idx = np.argsort(d)[:k]                  # indices of the k nearest points
    votes = np.bincount(y_train[idx])        # number of neighbours in each class
    return np.argmax(votes)
```

**Solution.** 0.75 for the distances, 0.75 for the $k$ nearest indices, 0.5 for the vote. Squared distances are fine since they have the same order.

**QUESTION 12** 1 pt

We increase $k$ from 1 to $m$ (the number of training points). Which statement is correct?

- ☐ **A** With $k = 1$ the training error is large, and it decreases when $k$ increases.
- ☐ **B** The boundary becomes more irregular when $k$ increases.
- ☐ **C** The value of $k$ has no effect once the features are standardised.
- ☑ **D** With $k = 1$ the model overfits. With $k = m$ it predicts the majority class of the train set everywhere.

**Solution.** $k$ controls the complexity: a small $k$ gives a flexible, unstable model, a large $k$ a smooth one. It is chosen on a validation set.

**QUESTION 13** 1 pt

An SVM with the Gaussian kernel $\kappa(f, f') = \exp(-\gamma\|f - f'\|^2)$ reaches 100 % accuracy on the train set and 55 % on the test set. What should you try?

- ☐ **A** Increase $\gamma$.
- ☑ **B** Decrease $\gamma$, choosing its value by cross-validation.
- ☐ **C** Remove the standardisation of the features.
- ☐ **D** Evaluate the model on the train set instead.

**Solution.** The model overfits. A large $\gamma$ gives a narrow kernel: each training point only influences its close neighbourhood, and the boundary can wrap around every point. A smaller $\gamma$ gives a smoother boundary.

## PART 5. TREES AND ENSEMBLES — 3 pts

**QUESTION 14** 1 pt

How are the trees of a **bagging** model trained?

- ☐ **A** Each tree is trained on the errors of the previous trees.
- ☑ **B** Each tree is trained independently, on a sample drawn with replacement from the train set.
- ☐ **C** Each tree is trained on the train set with weights that increase on misclassified points.
- ☐ **D** All trees are trained on the same data and differ only by their depth.

**Solution.** (A) describes gradient boosting and (C) AdaBoost.

**QUESTION 15** 1 pt

Which statement about the base trees is correct?

- ☑ **A** Boosting uses shallow trees to reduce the bias, bagging uses deep trees to reduce the variance.
- ☐ **B** Boosting uses deep trees to reduce the variance, bagging uses shallow trees to reduce the bias.
- ☐ **C** Both use stumps (trees of depth 1).
- ☐ **D** Both use fully grown trees.

**QUESTION 16** 1 pt

In a random forest, each split only considers a random subset of the features (parameter `max_features`). Why?

- ☐ **A** To select the most important features before training.
- ☐ **B** To limit the depth of the trees.
- ☐ **C** To give more weight to the misclassified points.
- ☑ **D** To make the trees more different from each other, so that averaging them reduces the variance more.

## NUMPY AND SCIKIT-LEARN REMINDER

| | |
|---|---|
| `A.shape` | tuple of dimensions, e.g. `(m, p)` |
| `A.T`, `A @ B` | transpose, matrix product |
| `A.mean(axis=0)` | mean of each column (shape `(p,)`) |
| `A.mean(axis=1, keepdims=True)` | mean of each row, kept as a column (shape `(m, 1)`) |
| `np.sum(A, axis=1)` | sum of each row |
| `np.eye(n)` | identity matrix of size $n$ |
| `np.zeros((n, n), dtype=int)` | $n \times n$ matrix of integer zeros |
| `for a, b in zip(u, v):` | loop over the pairs `(u[i], v[i])` |
| `np.diag(v)` | diagonal matrix with the vector `v` on its diagonal |
| `np.log(v)`, `np.exp(v)` | logarithm, exponential of each entry of `v` |
| `A.max(axis=1, keepdims=True)` | maximum of each row, kept as a column (shape `(m, 1)`) |
| `A.sum(axis=1, keepdims=True)` | sum of each row, kept as a column (shape `(m, 1)`) |
| `lam, U = np.linalg.eigh(S)` | eigenvalues `lam` (shape `(n,)`) and eigenvectors `U` (in columns) of a symmetric matrix `S`, with `S = U @ np.diag(lam) @ U.T` |
| `np.linalg.solve(A, b)` | solution `w` of the linear system `A @ w = b` |
| `i, j = np.triu_indices(n)` | row and column indices of the upper triangle (diagonal included) of an $n \times n$ matrix. `A[i, j]` is the vector of these entries |
| `np.argsort(v)` | indices that sort `v` in increasing order |
| `np.argmax(v)` | index of the largest entry of `v` |
| `np.bincount(v)` | for a vector of non-negative integers, number of occurrences of $0, 1, 2, \ldots$ |
| `v[idx]` | entries of `v` at the indices `idx` |
| `scaler = StandardScaler().fit(A)` | computes the mean and standard deviation of each column of `A` |
| `scaler.transform(A)` | standardises `A` with the stored mean and standard deviation |
| `scaler.fit_transform(A)` | `fit`, then `transform`, on the same data |
| `train_test_split(A, y, test_size=0.25)` | random split, returns `A_train, A_test, y_train, y_test` |
| `model.fit(A, y)`, `model.score(A, y)` | training, accuracy |