Predicting DPE Labels from Open Data

Model Build Report — Paris, 2018–2021

Open Data Challenge *(OpenData University × Enedis)* · PBL Year 2

Forward College

April 1, 2026

Contents

1. Overview

This report documents the complete design, training, and evaluation of a machine-learning classifier that predicts the DPE energy label (*étiquette DPE*, from **A** to **G**) from a deliberately small and interpretable set of building descriptors available in ADEME open data.

1.1 Motivation

The *Diagnostic de Performance Énergétique* (DPE) is a mandatory energy audit for residential and commercial properties in France. It assigns a label on a seven-class scale — A (most efficient) through G (least efficient) — based on primary energy consumption and greenhouse-gas emissions. Because DPE records are rich in structured metadata (surface area, construction era, heating fuel, etc.), a well-calibrated classifier can be used for at least three practical purposes:

(i) **Quality control.** Detecting records whose label appears inconsistent with the accompanying numerical fields — a useful pre-processing step before any downstream analysis.

(ii) **Imputation.** Filling in a missing label when all structured fields are present.

(iii) **Auxiliary signal.** Feeding probabilistic label estimates into a larger product pipeline alongside measured consumption data (Enedis).

1.2 Scope and Position in the Broader Project

The wider Open Data Challenge project connects DPE information to metered electricity consumption provided by Enedis (`consommation-annuelle-residentielle-par-adresse`). The present report focuses exclusively on the label-prediction sub-task. A future deliverable will document a regression model trained on measured consumption and augmented with prediction intervals, for which the classifier described here will serve as an interpretability anchor.

2. Data Sources and Scope

2.1 Geographic and Temporal Restriction

The analysis is restricted to **Paris** and the period **2018–2021**. This choice balances data volume (Paris has a large number of DPE records) with homogeneity (Parisian building stock is relatively uniform in construction type and climate zone).

2.2 ADEME DPE Datasets

The DPE methodology underwent a significant revision in July 2021. ADEME consequently distributes two distinct open datasets:

`dpe-france` Pre-2021 DPE records. These follow the old methodology, where the label was derived from either energy consumption or GHG emissions, whichever produced the worse rating (*double seuil*).

`dpe03existant` Post-July-2021 records. The revised methodology uses a single composite score weighting both primary energy and GHG emissions explicitly.

Because the two datasets use partially different variable names and label definitions, both were loaded and harmonised before union. Records straddling the methodological boundary (i.e. assessments performed in mid-2021) were excluded to avoid label noise.

2.3 Enedis Consumption Data

The Enedis dataset provides annual residential electricity consumption at address level. It was filtered to Parisian addresses and serves the regression sub-task only; it does not enter the classifier described here.

3. Target and Feature Design

3.1 Supervised Target

The target variable is

$$y \in \mathcal{Y} = \{A, B, C, D, E, F, G\},$$

encoded as integer class indices $\{0, 1, \ldots, 6\}$ for scikit-learn compatibility. This is a *multiclass*, *single-label* classification problem.

3.2 Raw Input Variables

Six raw variables are used, deliberately chosen to be available in both ADEME datasets and to cover the main physical determinants of a DPE label.

Table 1: Raw input variables before preprocessing

Variable	Type	Physical meaning
`surface_habitable_logement`	Numerical	Floor area of the dwelling (m^2)
`conso_5_usages_par_m2_ep`	Numerical	Standardised primary energy consumption per m^2 ($kWh_{ep}/m^2/yr$), aggregating the five regulated uses: heating, DHW, cooling, ventilation, lighting
`emission_ges_5_usages`	Numerical	GHG emissions for the same five uses (kg CO_2eq/m^2/yr)
`type_batiment`	Categorical	Building type (e.g. *maison*, *appartement*, *immeuble*)
`periode_construction`	Categorical	Construction period bracket (e.g. avant 1948, 1948–1974, ...)
`type_energie_principale_chauffage`	Categorical	Main heating fuel (e.g. gaz naturel, électricité, fioul)

Remark 3.1. The two most predictive variables are almost certainly `conso_5_usages_par_m2_ep` and `emission_ges_5_usages`, since the DPE label is computed directly from these quantities. Including them is therefore a mild form of *label leakage*: the model is partly reconstructing a deterministic function. This is acceptable here because the stated goal is quality-checking (detecting inconsistent records), not predicting labels for buildings that lack these fields. If the use-case changes, these variables should be re-evaluated.

3.3 Post-Preprocessing Dimensionality

After one-hot encoding the three categorical variables, the feature space becomes a 24-dimensional vector:

$$\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^{24}.$$

The 21 binary dimensions come from the one-hot expansion of the categorical variables (the exact cardinality of each depends on the number of observed categories in the training data).

4. Preprocessing Pipeline

4.1 Design Principle: Unified Pipeline

All transformations are encapsulated in a `scikit-learn` `Pipeline` object, ensuring that *exactly* the same operations are applied at training time and at inference time. This is critical: fitting a scaler on the test set (or re-fitting it during inference) would introduce data leakage or distribution mismatch.

4.2 Numerical Preprocessing

Step 1. Median imputation. Missing values in numerical columns are replaced by the median computed on the training set. The median is preferred over the mean because energy-consumption variables are right-skewed (a few extremely inefficient buildings pull the mean upward).

Step 2. StandardScaler (zero-mean disabled). Variables are divided by their training-set standard deviation but *not* centred (i.e. `with_mean=False`). This is appropriate when the data may be sparse after imputation, as centring would destroy sparsity. Formally, for numerical feature j:

$$\phi_j = \frac{x_j}{\hat{\sigma}_j},$$

where $\hat{\sigma}_j$ is the empirical standard deviation on the training split.

4.3 Categorical Preprocessing

Step 3. Mode imputation. Missing categories are replaced with the most frequent category observed in the training set.

Step 4. Type coercion. A custom `to_str` transformer ensures all values are Python `str` objects before encoding, preventing type errors when integer-coded categories (e.g. construction period codes) are mixed with string values.

Step 5. OneHotEncoder (ignore unknown). Each categorical variable is expanded into a binary indicator vector. Setting `handle_unknown="ignore"` means that categories unseen during training produce an all-zero vector at inference time, rather than raising an error.

4.4 Known Issue: Joblib Pickling of `to_str`

When the pipeline is saved with `joblib.dump` and reloaded in a different Python session, a common error arises:

Listing 1: Error when reloading the pipeline

```
AttributeError: Can't␣get␣attribute␣'to_str'
␣␣␣␣on␣<module␣'__main__'>
```

This happens because `to_str` was defined inside a Jupyter notebook (`__main__`), and `joblib` serialises a *reference* to that symbol rather than its bytecode. Upon loading in a new session, `__main__` no longer contains `to_str`.

Workaround (immediate): Define `to_str` in the loading script before calling `joblib.load`.

Robust fix: Move `to_str` into a standalone module (e.g. `preprocessing_utils.py`), import it during both training and inference, and re-save the pipeline. `joblib` will then serialise a reference to the module, which is stable across sessions.

5. Train/Test Split and Dataset Sizes

5.1 Stratified Shuffle Split

The dataset was split 80% training / 20% test using `StratifiedShuffleSplit` (`test_size=0.2`, `random_state=42`). Stratification ensures that the relative frequency of each label A–G is preserved in both splits. This matters because class A is extremely rare (43 test examples): a non-stratified split might place *all* A examples in the training set.

5.2 Development Subset

For rapid iteration, a development (*dev*) subset was constructed by sampling at most 15 000 observations per class from the training split. This caps the size while maintaining balance across classes.

Table 2: Dataset sizes after splitting

Split	Rows	Columns (raw)
Dev set	80 349	11
Test set	150 155	11

5.3 Leakage Risk from Duplicate Records

The current split is stratified but *not grouped*. If two DPE records refer to the same physical dwelling (e.g. a re-assessment of the same apartment), one copy may appear in the training set and another in the test set. Because the two records are nearly identical, the model effectively "sees" the test observation during training, inflating the apparent generalisation performance.

Remark 5.1. If duplicate records are non-negligible in volume, the split should be redone using `GroupShuffleSplit` or `GroupKFold` with the building/address identifier as the grouping key. This provides a more conservative and honest estimate of out-of-sample performance.

6. Model Choice and Training

6.1 Why Multinomial Logistic Regression?

The choice of model family is driven by three requirements: (i) the output should be a *probability distribution* over labels, not just a hard prediction; (ii) the model should be *interpretable* — we want to be able to explain a predicted label in terms of specific feature contributions; and (iii) training should be *scalable* to tens of thousands of examples with sparse, high-dimensional features.

Multinomial logistic regression satisfies all three. It is the canonical probabilistic linear classifier, its parameters (one weight per feature per class) admit direct interpretation, and stochastic gradient descent (SGD) makes it efficient even for large sparse inputs.

6.2 SGDClassifier Configuration

In scikit-learn, multinomial logistic regression trained by SGD is obtained via `SGDClassifier(loss="log_loss")`:

Listing 2: Classifier instantiation

```
from sklearn.linear_model import SGDClassifier

clf = SGDClassifier(
    loss          = "log_loss",    # multinomial logistic (softmax)
    alpha         = 5e-6,          # L2 regularisation strength
    class_weight  = "balanced",    # up-weight rare classes
    max_iter      = 200,           # passes over the training data
    tol           = 1e-4,          # convergence tolerance
    random_state  = 42,
    n_jobs        = -1,            # use all CPU cores
)
```

6.3 Loss Function: Cross-Entropy

For a single observation $(\mathbf{x}, y)$ with true class $y \in \mathcal{Y}$ and predicted probability vector $\hat{\mathbf{p}} = (\hat{p}_A, \ldots, \hat{p}_G)$, the cross-entropy loss is:

$$\mathcal{L}(\hat{\mathbf{p}}, y) \;=\; -\log \hat{p}_y \;=\; -\sum_{k \in \mathcal{Y}} \mathbf{1}[y = k] \, \log \hat{p}_k .$$

The loss is large when the model assigns low probability to the correct class, and small when it assigns high probability. Crucially, it is *unbounded from above*: a confident wrong prediction (e.g. $\hat{p}_y = 0.001$) incurs a loss of $\log(1000) \approx 6.9$, which applies strong gradient pressure to correct the error.

The total training loss is the mean over all training observations:

$$\mathcal{L}_{\text{train}} \;=\; \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{\mathbf{p}}^{(i)},\, y^{(i)}).$$

6.4 Regularisation: L2 Penalty

Without regularisation, gradient descent can push individual weights to extreme values to fit training noise, leading to overfitting. An L2 (ridge) penalty is added to the loss:

$$\mathcal{L}_{\text{total}} \;=\; \underbrace{\mathcal{L}_{\text{train}}}_{\text{fit the data}} \;+\; \underbrace{\alpha \sum_{k} \|\mathbf{w}_k\|^2}_{\text{penalise large weights}} \quad ,$$

where $\mathbf{w}_k \in \mathbb{R}^{24}$ is the weight vector for class k and $\alpha = 5 \times 10^{-6}$ controls the trade-off. A small α means the penalty is mild; the model is largely free to fit the data. This is appropriate here because with $n \approx 80\,000$ examples and only 175 parameters, the model is already heavily over-determined and strong regularisation is not needed.

Intuition. Think of α as a tax on complexity. A low tax (small α) barely changes behaviour. A high tax (large α) forces the model to keep weights small, producing a simpler decision boundary — potentially at the cost of underfitting.

6.5 Handling Class Imbalance: Balanced Weights

The class distribution is severely skewed (class A: 43 test examples; class D: 43 876). Without correction, the model converges toward predicting D for almost all inputs — achieving reasonable accuracy but failing entirely on rare classes.

Setting `class_weight="balanced"` assigns to each class k a weight inversely proportional to its frequency:

$$\omega_k \;=\; \frac{n}{K \cdot n_k},$$

where n is the total number of training examples, $K = 7$ is the number of classes, and n_k is the number of training examples belonging to class k. The loss for each observation is then multiplied by $\omega_{y^{(i)}}$, so the model incurs a larger penalty for misclassifying a rare class than a common one.

6.6 Stochastic Gradient Descent

Rather than computing the gradient of $\mathcal{L}_{\text{total}}$ over the entire training set at each step (which costs $O(n \cdot d)$ per step for d features), SGD approximates the gradient using a single randomly selected example (or a mini-batch):

$$\mathbf{w}_k \;\leftarrow\; \mathbf{w}_k \;-\; \eta_t \, \nabla_{\mathbf{w}_k} \mathcal{L}(\hat{\mathbf{p}}^{(i)}, y^{(i)}),$$

where η_t is the learning rate at step t (scikit-learn uses an adaptive schedule). Each full pass over the training data is called an *epoch* (or *iteration* in scikit-learn's terminology).

Non-convergence warning. Scikit-learn warned that the optimiser had not converged after `max_iter=200` epochs. This does not invalidate the model, but it means the weights have not reached a stationary point of the loss. Remedies include: increasing `max_iter`, tightening `tol`, or slightly increasing `alpha` to smooth the loss surface.

7. Mathematical Form of the Model

7.1 Notation

- $\mathbf{x} \in \mathbb{R}^6$: raw input vector (3 numerical + 3 categorical variables).
- $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^{24}$: preprocessed feature vector after the pipeline (scaling + one-hot encoding).
- $K = 7$: number of classes ($\mathcal{Y} = \{A, B, C, D, E, F, G\}$).
- $\mathbf{W} \in \mathbb{R}^{K \times 24}$: learned weight matrix (`coef_`).
- $\mathbf{b} \in \mathbb{R}^K$: learned intercept vector (`intercept_`).
- $w_{k,j}$: weight for class k on feature dimension j.
- b_k: intercept for class k.

7.2 Step 1 — Linear Scores

For each class $k \in \mathcal{Y}$, a linear score is computed:

$$s_k(\mathbf{x}) \;=\; b_k \;+\; \sum_{j=1}^{24} w_{k,j}\,\phi_j(\mathbf{x}) \;=\; b_k + \mathbf{w}_k^\top \boldsymbol{\phi}(\mathbf{x}). \tag{1}$$

This is a standard dot product between the k-th row of $\mathbf{W}$ and the feature vector, plus a bias. In matrix form, the vector of all scores is:

$$\mathbf{s}(\mathbf{x}) \;=\; \mathbf{b} + \mathbf{W}\,\boldsymbol{\phi}(\mathbf{x}) \;\in\; \mathbb{R}^7.$$

The scores are unbounded real numbers and have no direct probabilistic interpretation on their own.

7.3 Step 2 — Softmax Transformation

The scores are converted to a proper probability distribution via the *softmax* function:

$$P(y = k \mid \mathbf{x}) \;=\; \frac{\exp\Big(s_k(\mathbf{x})\Big)}{\displaystyle\sum_{c \in \mathcal{Y}} \exp\Big(s_c(\mathbf{x})\Big)}. \tag{2}$$

The softmax has three key properties:

(i) **Positivity.** $\exp(\cdot) > 0$ for all real inputs, so all probabilities are strictly positive.

(ii) **Normalisation.** The denominator ensures the probabilities sum to exactly 1.

(iii) **Monotonicity.** A higher score s_k always implies a higher probability $P(y = k \mid \mathbf{x})$, so the argmax of scores equals the argmax of probabilities.

Relationship to the sigmoid. For the binary case ($K = 2$, classes 0 and 1), the softmax reduces to the sigmoid:

$$P(y = 1 \mid \mathbf{x}) \;=\; \frac{e^{s_1}}{e^{s_0} + e^{s_1}} \;=\; \frac{1}{1 + e^{s_0 - s_1}} \;=\; \sigma(s_1 - s_0).$$

The sigmoid is therefore a *special case* of softmax for two classes. The current model ($K = 7$) uses the full softmax.

7.4 Step 3 — Prediction

The predicted label is the class with the highest probability (equivalently, the highest score, by monotonicity):

$$\hat{y}(\mathbf{x}) \;=\; \arg\max_{k \in \mathcal{Y}} \; s_k(\mathbf{x}) \;=\; \arg\max_{k \in \mathcal{Y}} \; P(y = k \mid \mathbf{x}). \tag{3}$$

7.5 Parameter Count and Model Size

The learned artifact has:

$$|\Theta| \;=\; K \times d + K \;=\; 7 \times 24 + 7 \;=\; \mathbf{175} \text{ parameters.}$$

This is extremely compact: the entire model can in principle be written as a spreadsheet, which is a strong interpretability advantage.

7.6 Interpreting the Weights: Relativity

A common misinterpretation is to read a negative weight $w_{k,j} < 0$ as meaning "feature j makes class k less likely." This is *not* generally true under softmax.

What matters for prediction is the *relative* score: whether $s_k > s_{k'}$ for all competing classes k'. Formally, consider two classes k and k':

$$s_k(\mathbf{x}) > s_{k'}(\mathbf{x}) \iff (b_k - b_{k'}) + (\mathbf{w}_k - \mathbf{w}_{k'})^\top \boldsymbol{\phi}(\mathbf{x}) > 0.$$

Only the *difference* in weights $(w_{k,j} - w_{k',j})$ for feature j determines whether that feature favours class k over k'.

Remark 7.1. This also implies an identification issue: adding the same constant vector $\mathbf{c}$ to all rows of $\mathbf{W}$ leaves every prediction unchanged. Scikit-learn implicitly handles this by the choice of optimiser, but it means the weight matrix has a $(K-1) \times d$ dimensional effective parameter space, not $K \times d$.

7.7 Worked Numerical Example

Suppose the preprocessed feature vector $\boldsymbol{\phi}(\mathbf{x})$ for a particular apartment yields the following scores (hypothetical):

$$\mathbf{s} = (-3.0,\; -1.0,\; 0.5,\; 1.2,\; 0.0,\; -2.0,\; -3.5) \quad \text{for classes } A, B, C, D, E, F, G.$$

Computing the softmax denominator:

$$Z = e^{-3.0} + e^{-1.0} + e^{0.5} + e^{1.2} + e^{0.0} + e^{-2.0} + e^{-3.5} \approx 0.050 + 0.368 + 1.649 + 3.320 + 1.000 + 0.135 + 0.030 = 6.552.$$

The resulting probability distribution is:

The model predicts **D** with 50.67% probability. Note that class C is also plausible (25.17%), providing useful uncertainty information that a hard classifier would discard.

Table 3: Softmax output for the worked example

Class	**Score** s_k	$\mathbf{e^{s_k}}$	**Probability (%)**
A	-3.0	0.050	0.76
B	-1.0	0.368	5.61
C	0.5	1.649	25.17
D	**1.2**	**3.320**	**50.67**
E	0.0	1.000	15.26
F	-2.0	0.135	2.06
G	-3.5	0.030	0.46
Sum		6.552	100.00

8. Evaluation Results

8.1 Overall Metrics

Evaluation was performed on the held-out test set ($n = 150\,155$).

Table 4: Summary of overall test-set performance

Metric	Value
Accuracy	0.730
Balanced accuracy (macro recall)	0.798
Macro-averaged F1	0.720
Weighted F1	0.720

The gap between accuracy (0.73) and balanced accuracy (0.798) is informative: accuracy is dragged down by errors on the numerically large classes (C, D), while balanced accuracy treats each class equally and reveals that the model does better on rare classes than raw accuracy suggests.

8.2 Per-Class Results

Table 5: Per-class precision, recall, F1, and support on the test set

Class	Precision	Recall	F1	Support
A	—	—	—	43
B				
C			0.85	25 859
D			0.71	43 876
E	0.89	0.45		
F				
G				
Macro avg			0.72	150 155
Weighted avg			0.72	150 155

Class A. With only 43 test examples, precision and recall estimates are statistically unreliable. A single misclassification changes recall by more than 2 percentage points. No conclusions should be drawn from class-A metrics alone.

Classes C and D. These dominate the dataset and are predicted relatively well (F1 of 0.85 and 0.71 respectively).

Class E. The high precision (0.89) but low recall (0.45) pattern indicates the model is *conservative* about predicting E. When it does predict E, it is almost always right, but it misses roughly half of all true E examples, classifying them as D or F instead. This is a symptom of the decision boundary being poorly calibrated around E — likely because E neighbours both D and F and the training signal is ambiguous in that region.

9. Limitations and Next Improvements

9.1 Technical Improvements

Non-convergence. Increase `max_iter` (e.g. to 1000) and monitor the training loss curve. Alternatively, switch to `LogisticRegression` with `solver="saga"`, which is also SGD-based but better supported for multiclass problems.

Grouped split. If duplicate DPE records per building are non-negligible, re-evaluate using a grouped split keyed on building/address identifier to obtain a conservative out-of-sample estimate.

Hyperparameter search. The current `alpha` was not tuned via cross-validation. A grid search over $\alpha \in \{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}\}$ would confirm whether the current choice is close to optimal.

9.2 Modelling Improvements

Class A scarcity. With 43 test examples, it is not possible to reliably evaluate or optimise performance on class A. Depending on product needs, merging A and B into a single "high efficiency" band may be more stable.

Ordinal structure. The DPE scale A–G is ordinal. A misclassification of E as D is less severe than a misclassification of E as A. Ordinal regression (e.g. proportional odds logistic regression) or a custom cost-sensitive loss would encode this structure explicitly.

Non-linear models. A gradient-boosted tree (e.g. LightGBM) would likely improve accuracy, particularly for class E where the linear decision boundary appears poorly calibrated. The trade-off is reduced interpretability.

9.3 Roadmap Toward the Reality-Adjusted DPE

The next deliverable in the project is a regression model trained on measured Enedis electricity consumption. The classifier described here will serve as:

- An auxiliary consistency signal (does the predicted label match the measured consumption?).
- An interpretability anchor for the regression output.
- A feature (label probabilities) that can be fed into the regression model directly.

Prediction intervals (e.g. quantile regression or conformal prediction) will be added to the regression model to communicate uncertainty to end-users.

Appendix

A. Scikit-learn Pipeline Pseudocode

Listing 3: Full pipeline definition

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

from preprocessing_utils import to_str   # avoid __main__
   pickling issue

NUMERICAL_FEATURES = [
    "surface_habitable_logement",
    "conso_5_usages_par_m2_ep",
    "emission_ges_5_usages",
]

CATEGORICAL_FEATURES = [
    "type_batiment",
    "periode_construction",
    "type_energie_principale_chauffage",
]

numerical_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler",  StandardScaler(with_mean=False)),
])

categorical_transformer = Pipeline([
    ("imputer",  SimpleImputer(strategy="most_frequent")),
    ("to_str",   FunctionTransformer(to_str)),
    ("encoder",  OneHotEncoder(handle_unknown="ignore")),
])

preprocessor = ColumnTransformer([
    ("num", numerical_transformer,   NUMERICAL_FEATURES),
    ("cat", categorical_transformer, CATEGORICAL_FEATURES),
])

pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier",   SGDClassifier(
        loss          = "log_loss",
        alpha         = 5e-6,
        class_weight  = "balanced",
        max_iter      = 200,
```

```
        tol           = 1e-4,
        random_state  = 42,
        n_jobs        = -1,
    )),
])
```

B. Softmax vs. Sigmoid: Formal Equivalence

For $K = 2$ with scores s_0 and s_1:

$$\begin{aligned} P(y = 1 \mid \mathbf{x}) &= \frac{e^{s_1}}{e^{s_0} + e^{s_1}} \\ &= \frac{1}{e^{s_0 - s_1} + 1} \\ &= \sigma(s_1 - s_0), \end{aligned}$$

where $\sigma(z) = (1 + e^{-z})^{-1}$ is the sigmoid function. The binary cross-entropy loss is a special case of the multiclass cross-entropy, confirming that softmax regression strictly generalises binary logistic regression.