01-03 ML fondementals _Validation

Terminologies:

- **Generalization**: generalize from what we've seen to what we haven't seen
- Features, target, examples, training
- **Supervised vs. Unsupervised machine learning**
  - In **supervised learning**, training data comprises a set of features (X) and their corresponding targets (y). We wish to find a **model function f** that relates X to y. Then use that model function **to predict the targets** of new examples.
  - In **unsupervised learning** training data consists of observations (X) **without any corresponding targets**. Unsupervised learning could be used to **group similar things together** in X or to provide **concise summary** of the data.
- **Classification and regression**
  - **Classification problem**: predicting among two or more discrete classes
    - Example1: Predict whether a patient has a liver disease or not
    - Example2: Predict whether a student would get an A+ or not in quiz2.
  - **Regression problem**: predicting a continuous value
    - Example1: Predict housing prices
    - Example2: Predict a student's score in quiz2.
- **Accuracy** (.score() for classification problems)
- **error**: 1 - accuracy
- **Parameters and hyperparameters**
  - **Parameters** : When you call `fit`, a bunch of values get set, like the features to split on and split thresholds. These are called **parameters**. These are learned by the algorithm from the data during training. We need them during prediction time.
  - **Hyperparameters** : Even before calling `fit` on a specific data set, we can set some "knobs" that control the learning. These are called **hyperparameters**. These are specified based on: expert knowledge, heuristics, or systematic/automated optimization (more on this in the coming lectures).
- **Decision boundary**
  - the X-dimensional boundaries among the predicted results.
- **Decision Trees**
  - **A root node** : represents the first condition to check or question to ask
  - **A branch** : connects a node (condition) to the next node (condition) in the tree. Each branch typically represents either true or false.
  - **An internal node** : represents conditions within the tree

- **A leaf node** : represents the predicted class/value when the path from root to the leaf node is followed.
- **Tree depth** : The number of edges on the path from the root node to the farthest away leaf node.
- A decision tree with only one split (depth=1) is called a **decision stump.**
- `max_depth` is a **hyperparameter** of `DecisionTreeClassifier`.

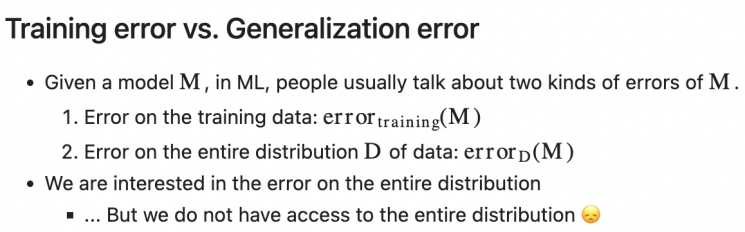
How does prediction work

- In summary, given a learned tree and a test example, during prediction time,
- Start at the top of the tree. Ask binary questions at each node and follow the appropriate path in the tree. Once you are at a leaf node, you have the prediction.
- Note that the model only considers the features which are in the learned tree and ignores all other features.

How does fit work

- Fitting a Decision tree for classification problems
  - Which features are most useful for classification?
  - Minimize **impurity** at each question
  - Common criteria to minimize impurity: gini index, information gain, cross entropy
- Fitting Decision tree for regression problems
  - Instead of gini, we use some other criteria for splitting. A common one is mean squared error (MSE). (More on this in later videos.)

Test Error: how well the model can be generalized



Data splitting approximate generalization error / test error

Cross validation

- It creates `cv` folds on the data.
- In each fold, it fits the model on the training portion and scores on the validation portion.
- The output is a list of validation scores in each fold.

Advantages of cross validation

- **Optimal Hyperparameter Tuning:** By evaluating the model on multiple subsets of the data, cross-validation helps in selecting the hyperparameters that generalize well. hyperparameter tuning (e.g., grid search or random search)
- **Bias and Variance Assessment:** helps in diagnosing issues related to bias (underfitting) and variance (overfitting)
- **Robust model assessment:** Provides a more robust estimate of a model's performance compared to a single train-test split. as it helps reduce the impact of the randomness in the data partitioning process.
- **Maximizing Data Utilization: In k-fold cross-validation, you train and test the model k times, using different subsets of the data. This means you get to use nearly all your data for training and testing.**

Underfitting

- If your model is too simple, like `DummyClassifier` or `DecisionTreeClassifier` with `max_depth=1`, it's not going to pick up on some random quirks in the data but it won't even capture useful patterns in the training data.
- The model won't be very good in general. Both train and validation errors would be high. This is **underfitting**.
- The gap between train and validation error is going to be lower.

$$E_{best} < E_{train} \lesssim E_{valid}$$

Overfitting

- If your model is very complex, like a `DecisionTreeClassifier(max_depth=None)`, then you will learn unreliable patterns in order to get every single training example correct.
- The training error is going to be very low but there will be a big gap between the training error and the validation error. This is **overfitting**.
- In overfitting scenario, usually we'll see:

$$E_{train} < E_{best} < E_{valid}$$

The "fundamental tradeoff" of supervised learning:

As you increase model complexity, Training Error tends to go down but difference between the Train error and Valid Error tends to go up.

The fundamental trade-off is also called the bias/variance tradeoff in supervised machine learning.

Bias: the tendency to consistently learn the same wrong thing (high bias corresponds to underfitting)

Variance: the tendency to learn random things irrespective of the real signal (high variance corresponds to overfitting)

The golden rule

- Even though we care the most about test error **THE TEST DATA CANNOT INFLUENCE THE TRAINING PHASE IN ANY WAY**.
- We have to be very careful not to violate it while developing our ML pipeline.
- Even experts end up breaking it sometimes which leads to misleading results and lack of generalization on the real data.

04 KNN & SVM

Pros of k-NNs for supervised learning

- Easy to understand, interpret.
- Simple hyperparameter k (`n_neighbors`) controlling the fundamental tradeoff.
- Can learn very complex functions given enough data.
- Lazy learning: Takes no time to `fit`

Cons of k-NNs for supervised learning

- Can be potentially be VERY slow during prediction time, especially when the training set is very large.
- Often not that great test accuracy compared to the modern approaches.
- It does not work well on datasets with many features or where most feature values are 0 most of the time (sparse datasets).

```
:class: important

For regular $k$-NN for supervised learning (not with sparse
matrices), you should scale your features. We'll be looking into
it soon.
```

Support Vector Machines (SVMs) with RBF kernel

- Superficially, SVM RBFs are more like weighted kNNs.
  - The decision boundary is defined by **a set of positive and negative examples** and **their weights** together with **their similarity measure**.
  - A test example is labeled positive if on average it looks more like positive examples than the negative examples.

  unlike decision trees, all features are equally important
- The primary difference between NNs and SVM RBFs is that
  - Unlike kNNs, SVM RBFs only remember the key examples (support vectors).
  - SVMs use a different similarity metric which is called a "kernel". A popular kernel is Radial Basis Functions (RBFs)
  - They usually perform better than kNNs!

Hyperparameters of SVM

gmma and C controls the complexity (fundamental trade-off), just like other hyperparameters we've seen.

Larger gamma → more complex

larger C → more complex

05-06 Processing, pipelines, columnTransformer and Text Features

Common preprocessing techniques

Some commonly performed feature transformation include:

- Imputation: Tackling missing values
- Scaling: Scaling of numeric features
- One-hot encoding: Tackling categorical variables

risks of applying transformers in cross validations: allowing information validation set to leak into the training step

Use sklearn pipeline

Categorical features:

One-hot encoding (OHE)

- Create new binary columns to represent our categories.
- If we have c categories in our column.
  - We create c new binary columns to represent those categories.
- Example: Imagine a language column which has the information on whether you
- We can use sklearn's `OneHotEncoder` to do so.

handling unknows in ohe

- Pass `handle_unknown="ignore"` argument to `OneHotEncoder`: It creates a row with all zeros.
- **'handle_unknown = error' (default)**
- **'handle_unknown = use_encoded_value'**

Cases where it's OK to break the golden rule

- If it's some fix number of categories. For example, if it's something like provinces in Canada or majors taught at UBC. We know the categories in advance and this is one of the cases where it might be OK to violate the golden rule and get a list of all possible values for the categorical variable.

Categorical features with only two possible categories

- Sometimes you have features with only two possible categories.
- If we apply `OheHotEncoder` on such columns, it'll create two columns, which seems wasteful, as we could represent all information in the column in just one column with say 0's and 1's with presence of absence of one of the categories.

- You can pass `drop="if_binary"` argument to `OneHotEncoder` in order to create only one column in such scenario.

OHE with many categories

- Do we have enough data for rare categories to learn anything meaningful?
- How about grouping them into bigger categories?
  - Example: country names into continents such as "South America" or "Asia"
- Or having "other" category for rare cases?

`OneHotEncoder` and sparse features

- By default, `OneHotEncoder` also creates sparse features.
- You could set `sparse=False` to get a regular `numpy` array.
- If there are a huge number of categories, it may be beneficial to keep them sparse.
- For smaller number of categories, it doesn't matter much.

07 Linear regression

`Ridge`

- `scikit-learn` has a model called `LinearRegression` for linear regression.
- But if we use this "vanilla" version of linear regression, it may result in large coefficients and unexpected results.
- So instead of using `LinearRegression`, we will always use another linear model called `Ridge`, which is a linear regression model with a complexity hyperparameter `alpha`.

Hyperparameter `alpha` of `Ridge` ¶

- Ridge has hyperparameters just like the rest of the models we learned.
- The alpha hyperparameter is what makes `Ridge` different from vanilla `LinearRegression`.
- Similar to the other hyperparameters that we saw, `alpha` controls the fundamental tradeoff.

> If we set alpha=0 that is the same as using LinearRegression.

Main hyperparameter of logistic regression

- `C` is the main hyperparameter which controls the fundamental trade-off.
- We won't really talk about the interpretation of this hyperparameter right now.
- At a high level, the interpretation is similar to `C` of SVM RBF
  - smaller `C` $\rightarrow$ might lead to underfitting
  - bigger `C` $\rightarrow$ might lead to overfitting

Components of a linear classifier

1. Input features $(x_1, \ldots, x_d)$
2. Coefficients (weights) $(w_1, \ldots, w_d)$
3. Bias (b or w_0) (can be used to offset your hyperplane)
4. Threshold (r)

08 Hyperparameter Optimization

Picking good hyperparameters is important because if we don't do it, we might end up with an underfit or overfit model.

Some ways to pick hyperparameters:

- Manual or expert knowledge or heuristics based optimization
- Data-driven or automated optimization

GridSearchCV

`n_jobs=-1`

- Note the `n_jobs=-1` above.
- Hyperparameter optimization can be done *in parallel* for each of the configurations.
- This is very useful when scaling up to large numbers of machines in the cloud.
- When you set `n_jobs=-1`, it means that you want to use all available CPU cores for the task.

The `__` syntax

- Above: we have a nesting of transformers.
- We can access the parameters of the "inner" objects by using __ to go "deeper":
- `svc__gamma`: the `gamma` of the `svc` of the pipeline
- `svc__C`: the `C` of the `svc` of the pipeline
- `columntransformer__countvectorizer__max_features`: the `max_features` hyperparameter of `CountVectorizer` in the column transformer `preprocessor`.

Problems with exhaustive grid search

- Required number of models to evaluate grows exponentially with the dimensionally of the configuration space.
- Example: Suppose you have
  - 5 hyperparameters
  - 10 different values for each hyperparameter
  - You'll be evaluating models! That is you'll be calling `cross_validate` 100,000 times!

    105=100,000
- Exhaustive search may become infeasible fairly quickly.

- Other options?

RandomizedSearchCV

- Faster compared to `GridSearchCV`.
- Adding parameters that do not influence the performance does not affect efficiency.
- Works better when some parameters are more important than others.

n_iter*int, default=10*

Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

Optimization bias/Overfitting of the validation set

Overfitting of the validation error

- Why do we need to evaluate the model on the test set in the end?
- Why not just use cross-validation on the whole dataset?
- While carrying out hyperparameter optimization, we usually try over many possibilities.
- If our dataset is small and if your validation set is hit too many times, we suffer from **optimization bias** or **overfitting the validation set**.

Optimization bias of parameter learning

- Overfitting of the training error
- An example:
  - During training, we could search over tons of different decision trees.
  - So we can get "lucky" and find a tree with low training error by chance.

Optimization bias of hyper-parameter learning

- Overfitting of the validation error
- An example:
  - Here, we might optimize the validation error over 1000 values of `max_depth`.
  - One of the 1000 trees might have low validation error by chance.

Large datasets solve many of these problems

- With infinite amounts of training data, overfitting would not be a problem and you could have your test score = your train score.
  - Overfitting happens because you only see a bit of data and you learn patterns that are overly specific to your sample.
  - If you saw "all" the data, then the notion of "overly specific" would not apply.
- So, more data will make your test score better and robust.

09 classification metrics

Precision: Among the positive examples you identified, how many were actually positive?

Recall: Among all positive examples, how many did you identify correctly?

F1 score:

- F1-score combines precision and recall to give one score, which could be used in hyperparameter optimization, for instance.
- F1-score is a harmonic mean of precision and recall.

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN} = \frac{TP}{\#positives}$$

$$f1 = 2 \times \frac{precision \times recall}{precision + recall}$$

Precision/Recall tradeoff

- But there is a trade-off between precision and recall.
- If you identify more things as "fraud", recall is going to increase but there are likely to be more false positives.

Decreasing the threshold

- Decreasing the threshold means a lower bar for predicting fraud.
  - You are willing to risk more false positives in exchange of more true positives.
  - Recall would either stay the same or go up and precision is likely to go down
  - Occasionally, precision may increase if all the new examples after decreasing the threshold are TPs.

Increasing the threshold

- Increasing the threshold means a higher bar for predicting fraud.
  - Recall would go down or stay the same but precision is likely to go up
  - Occasionally, precision may go down if TP decrease but FP do not decrease.

Precision-recall curve

- Confusion matrix provides a detailed break down of the errors made by the model.
- But when creating a confusion matrix, we are using "hard" predictions.
- Most classifiers in `scikit-learn` provide `predict_proba` method (or `decision_function`) which provides degree of certainty about predictions by the classifier.

- Can we explore the degree of uncertainty to understand and improve the model performance?

Change the threshold of the predict_proba

AP score

- Often it's useful to have one number summarizing the PR plot (e.g., in hyperparameter optimization)
- One way to do this is by computing the area under the PR curve.
- This is called **average precision** (AP score)
- AP score has a value between 0 (worst) and 1 (best).

AP vs. F1-score

It is very important to note this distinction:

- F1 score is for a given threshold and measures the quality of `predict`.
- AP score is a summary across thresholds and measures the quality of `predict_proba`.

Different classifiers might work well in different parts of the curve, i.e., at different operating points.

Receiver Operating Characteristic (ROC) curve

Similar to PR curve, it considers all possible thresholds for a given classifier given by `predict_proba` but instead of precision and recall it plots false positive rate (FPR) and true positive rate (TPR or recall).

- TPR : Fraction of true positives out of all positive examples.
- FPR : Fraction of false positives out of all negative examples.

$$TPR = \frac{TP}{TP + FN}$$
$$FPR = \frac{FP}{FP + TN}$$

- Different points on the ROC curve represent different classification thresholds. The curve starts at (0,0) and ends at (1, 1).
  - (0, 0) represents the threshold that classifies everything as the negative class
  - (1, 1) represents the threshold that classifies everything as the positive class
- The ideal curve is close to the top left
  - Ideally, you want a classifier with high recall while keeping low false positive rate.
- The red dot corresponds to the threshold of 0.5, which is used by predict.
- We see that compared to the default threshold, we can achieve a better recall of around 0.8 without increasing FPR.

Area under curve

AUC provides a single meaningful number for the model performance.

- AUC of 0.5 means random chance.
- AUC can be interpreted as evaluating the **ranking** of positive examples.
- What's the probability that a randomly picked positive point has a higher score according to the classifier than a randomly picked point from the negative class.
- AUC of 1.0 means all positive points have a higher score than all negative points.
- **For classification problems with imbalanced classes, using AP score or AUC is often much more meaningful than using accuracy.**

Class imbalance

Class imbalance in training sets

- This typically refers to having many more examples of one class than another in one's training set.
- Real world data is often imbalanced.
  - Our Credit Card Fraud dataset is imbalanced.
  - Ad clicking data is usually drastically imbalanced. (Only around ~0.01% ads are clicked.)
  - Spam classification datasets are also usually imbalanced.

Which type of error is more important?

- False positives (FPs) and false negatives (FNs) have quite different real-world consequences.
- In PR curve and ROC curve, we saw how changing the prediction threshold can change FPs and FNs.
- We can then pick the threshold that's appropriate for our problem.
- Example: if we want high recall, we may use a lower threshold (e.g., a threshold of 0.1). We'll then catch more fraudulent transactions.

handle class imbalance

Change the training procedure

All `sklearn` classifiers have a parameter called `class_weight`.

- This allows you to specify that one class is more important than another.
- For example, maybe a false negative is 10x more problematic than a false positive.
-

LogisticRegression(max_iter=500, class_weight={0: 1, 1: 10})

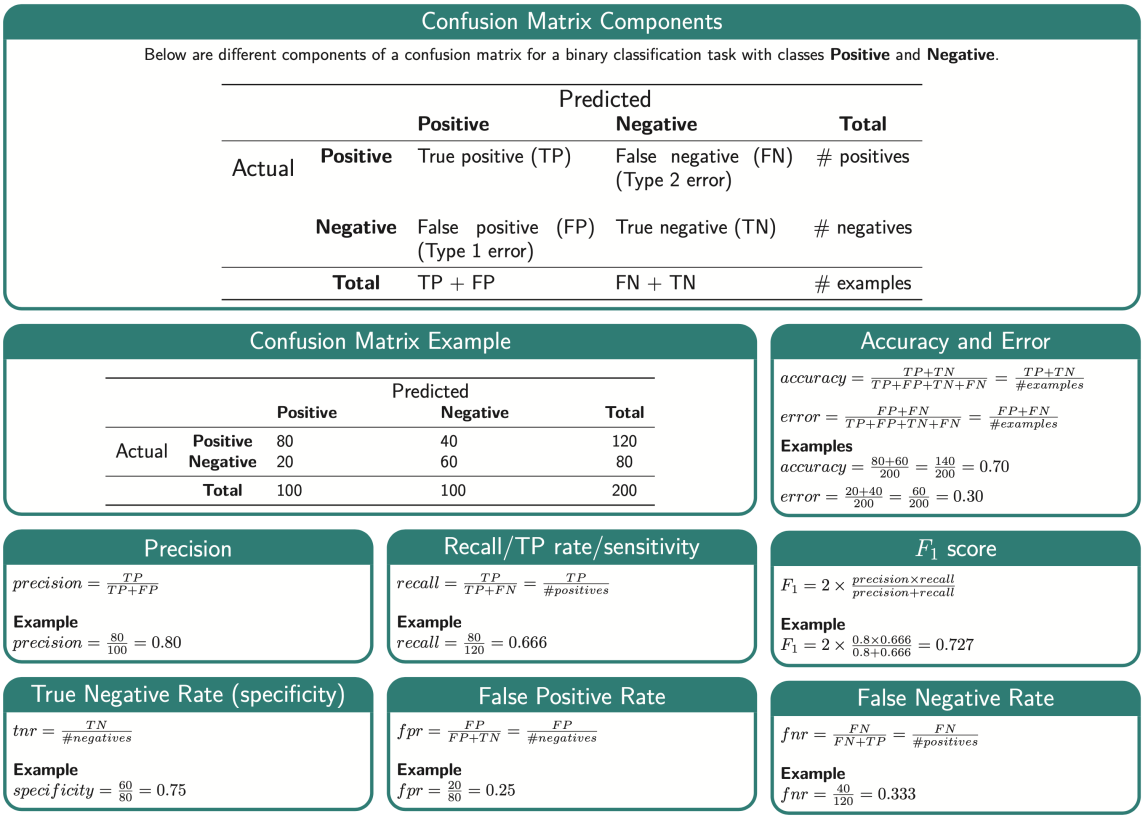
Class_weight allows you to specify one class is more important than another

class_weightdict or ‘balanced’, default=None

Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one.

handle class imbalance

This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.



10 Regression metrics

Ridge is a liner regression model with a complexity hyperparameter **alpha**, higher alpha means less complexity and is likely to underfit the model; Likewise, lower alpha means more complexity and is likely to overfit the model.

- General intuition: **larger alpha leads to smaller coefficients.**
- Smaller coefficients mean the predictions are less sensitive to changes in the data. Hence less chance of overfitting.

Coefficients and intercept (bias)

.coef_

.intercept_

Magnitude of the coefficients means the level of impact of the coefficients

Scaling

When you are interpreting the model coefficients, scaling is crucial.

- If you do not scale the data, features with smaller magnitude are going to get coefficients with bigger magnitude whereas features with bigger scale are going to get coefficients with smaller magnitude.
- That said, when you scale the data, feature values become hard to interpret for humans!

RidgeCV: automatically tunes `alpha` based on cross-validation

Mean Squared Error: perfect prediction has MSE = 0

The score depends on the scale of the target so the value, which maybe an unreasonable value

R^2 is the default .score(), it is unitless, 0 is bad, 1 is best

- The maximum is 1 for perfect predictions
- Negative values are very bad: "worse than DummyRegressor" (very bad)

Rooted mean square error or RMSE: the root of MS

MAPE: mean absolute percent error, more interpretable

Transforming the targets

Transform the target to be normal can increase the accuracy of the linear regression model

MSE (mean squared error) is in units of target squared, hard to interpret; 0 is best

RMSE (root mean squared error) is in the same units as the target; 0 is best

MAPE (mean absolute percent error) is unitless; 0 is best, 1 is bad

11 ensembles

Random Forest: train different decision trees, and vote for the prediction

- A single decision tree is likely to overfit
- Use a collection of diverse decision trees
- Each tree overfits on some part of the data but we can reduce overfitting by averaging the results
- can be shown mathematically

Inject randomness in the classifier construction

predict by voting (classification) or averaging(regression) of predictions given by individual models.

To ensure that the trees in the random forest are different we inject randomness in two ways:

1. Data: **Build each tree on a bootstrap sample** (i.e., a sample drawn **with replacement** from the training set)
2. Features: **At each node, select a random subset of features** (controlled by `max_features` in `scikit-learn`) and look for the best possible test involving one of these features

Hyperparameters:

n_estimators: how many decision trees we want to build

- `n_estimators`: number of decision trees (higher = more complexity)
- `max_depth`: max depth of each decision tree (higher = more complexity)

- `max_features`: the number of features you get to look at each split (higher = more complexity)

fit on a diverse set of that n decision trees by injecting randomness in the model construction

Strengths

- Usually one of the best performing off-the-shelf classifiers without heavy tuning of hyperparameters
- Don't require scaling of data
- Less likely to overfit
- In general, able to capture a much broader picture of the data compared to a single decision tree.

Weaknesses

- **Slower than decision trees because we are fitting multiple trees but can easily parallelize training because all trees are independent of each other**
- requires more memory
- hard to interpret
- tend not to perform well on high dimensional sparse data such as text data

Gradient boosted trees: Another popular and effective class of tree-based model, train models in sequences

- No randomization.
- **The key idea is combining many simple models called weak learners to create a strong learner.**
- They combine **multiple shallow (depth 1 to 5) decision trees.**
- They build trees in **a serial manner,** where each tree tries to correct the mistakes of the previous one.

XGBoost

- speed
- Supports missing values
- GPU training, networked parallel training
- Supports sparse data
- Typically better scores than random forests

LightGBM

- pros:
  - Small model size
  - Faster
  - Typically better scores than random forests

CatBoost

- Usually better scores but slower compared to `XGBoost` and `LightGBM`
- `n_estimators` Number of boosting rounds
- `learning_rate` The learning rate of training

        - controls how strongly each tree tries to correct the mistakes of the previous trees
        - higher learning rate means each tree can make stronger corrections, which means more complex model
    - `max_depth` `max_depth` of trees (similar to decision trees)
    - `scale_pos_weight` Balancing of positive and negative weights

methods to combine different models

Averaging

This `VotingClassifier` will take a *vote* using the predictions of the constituent classifier pipelines.

Main parameter: `voting`

- `voting='hard'`
    - it uses the output of `predict` and actually votes.
- `voting='soft'`
    - with `voting='soft'` it averages the output of `predict_proba` and then thresholds / takes the larger.

Stacking

- Another type of ensemble is stacking.
- Instead of averaging the outputs of each estimator, use their outputs as *inputs to another model*.
- By default for classification, it uses logistic regression.
    - We don't need a complex model here necessarily, more of a weighted average.
    - The features going into the logistic regression are the classifier outputs, *not* the original features!
    - So the number of coefficients = the number of base estimators!

Pick the best models: CV, interpretability, speed/code maintenance=

Although the most common approach is choosing the model which results in the **best mean cross-validation score** because it's easy for automation, choosing parameter values at "the sweet spot" where there is the least divergence between training and validation accuracies is a reasonable approach.

- the cross-validation score is close to the best cross-validation score
- the gap between train and cross-validation is not that large
- there is not much variation in the cross-validation sub-scores
- the model is simpler (less complex)

What is an advantage of ensembling multiple models as opposed to just choosing one of them?

- You may get a better score.

What is a disadvantage of ensembling multiple models as opposed to just choosing one of them?

- Slower, more code maintenance issues.

12 Feature importance

Termnologies

- feature_importance
  - Linear regression: the coefficients of each feature
- permutation_importance

SHAP:SHapley Additive exPlanations

SHAP values interpret the impact of having a certain value for a given feature in comparison to the prediction we'd make if that feature took some baseline value.

- Based on the idea of shapely values. A shapely value is created for each example and each feature.
- Can explain the prediction of an example by computing the contribution of each feature to the prediction.
- Great visualizations
- Support for different kinds of models; fast variants for tree-based models

```
shap.force_plot(
    lgbm_explainer.expected_value[1], # expected value for class 1.
    test_lgbm_shap_values[1][ex_l50k_index, :], # SHAP values associated with the example we want to explain
    X_test_enc.iloc[ex_l50k_index, :], # Feature vector of the example
    matplotlib=True,
)
```

Why model transparency/interpretability?

- Ability to interpret ML models is crucial in many applications such as banking, healthcare, and criminal justice.
- It can be leveraged by domain experts to diagnose systematic errors and underlying biases of complex ML systems.

What is model interpretability?

In this lecture, our definition of model interpretability will be looking at **feature importances**, i.e., exploring features which are important to the model.

13. Feature engineering and feature selection

Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data.

Better features usually help more than a better model.

- Good features would ideally:
  - capture most important aspects of the problem
  - allow learning with few examples
  - generalize to new scenarios.
- There is a trade-off between simple and expressive features:
  - With simple features overfitting risk is low, but scores might be low.
  - With complicated features scores can be high, but so is overfitting risk.

Feature selection

- With so many ways to add new features, we can increase dimensionality of the data.
- More features means more complex models, which means increasing the chance of overfitting.

Why feature selection?

- Interpretability: Models are more interpretable with fewer features. If you get the same performance with 10 features instead of 500 features, why not use the model with smaller number of features?
- Computation: Models fit/predict faster with fewer columns.
- Data collection: What type of new data should I collect? It may be cheaper to collect fewer columns.
- Fundamental tradeoff: Can I reduce overfitting by removing useless features?

Feature selection can often result in better performing (less overfit), easier to understand, and faster model.

Methods to conduct feature selection

Model-based selection

- Use a supervised machine learning model to judge the importance of each feature.
- Keep only the most important once.
- Supervised machine learning model used for feature selection can be different that the one used as the final estimator.
- Use a model which has some way to calculate feature importances.

RFE algorithm

1. Decide k, the number of features to select.
2. Assign importances to features, e.g. by fitting a model and looking at `coef_` or `feature_importances_`.

3. Remove the least important feature.
4. Repeat steps 2-3 until only k features are remaining.

Note that this is **not** the same as just removing all the less important features in one shot!

- Slow because there is cross validation within cross validation
- Not a big improvement in scores compared to all features on this toy case

Forward or backward selection

- Also called wrapper methods
- Shrink or grow feature set by removing or adding one feature at a time
- Makes the decision based on whether adding/removing the feature improves the CV score or not

Warnings about feature selection

- A feature's relevance is only defined in the context of other features.
  - Adding/removing features can make features relevant/irrelevant.
- If features can be predicted from other features, you cannot know which one to pick.
- Relevance for features does not have a causal relationship.
- Don't be overconfident.
  - The methods we have seen probably do not discover the ground truth and how the world really works.
  - They simply tell you which features help in predicting y_i for the data you have.