# Ensemble methods

- We were talking about a machine learning algorithm $\mathcal{A}$ creating a regression or classification model $f$ from a dataset $\mathcal{D}$
- **Ensemble models** create a series of models $f_1, f_2, \ldots$
  - Obviously, either the $\mathcal{D}$ has to vary in some way
  - Or the $\mathcal{A}$ should vary (slightly different parameters)
  - Or some kind of randomness should be involved in the learning.
- The models of the ensemble are combined in some ways to create a new, more powerful model $f$
- They are the most powerful models currently known, usually win competitions, etc.
  - Arguments were made that the human brain is an ensemble model

# Remember the expected error?

$$\underbrace{\mathbb{E}_{\mathcal{D}\sim P^n,(\boldsymbol{x},y)\sim P}\left[\left(f_{\mathcal{D}}(\boldsymbol{x})-y\right)^2\right]}_{\text{Expected Test Error}}=$$

$$\underbrace{\mathbb{E}_{\boldsymbol{x},\boldsymbol{\mathcal{D}}}\left[\left(f_{\mathcal{D}}(\boldsymbol{x})-\overline{f}(\boldsymbol{x})\right)^2\right]}_{\text{Variance}}+$$

$$\underbrace{\mathbb{E}_{\boldsymbol{x},y}\left[(\overline{y}(\boldsymbol{x})-y)^2)\right]}_{\text{Noise}}+$$

$$\underbrace{\mathbb{E}_{\boldsymbol{x}}\left[(\overline{f}(\boldsymbol{x})-\overline{y}(\boldsymbol{x}))^2\right]}_{\text{Bias}}$$

# Goal: reduce the variance

- Let us try to reduce the variance term $\mathbb{E}_{\boldsymbol{x},\boldsymbol{\mathcal{D}}}\left[\left(f_{\mathcal{D}}(\boldsymbol{x})-\overline{f}(\boldsymbol{x})\right)^2\right]$
- We want $f_{\mathcal{D}} \to \overline{f}$
  - The model learned on $D$ should get close to the average model
- Average of random samples converges to the mean! (weak law of large numbers)
- One way to achieve this is:
  - Pick random training data sets $\mathcal{D}_i \sim P, i = 1 \ldots m$
  - Train a model on them $f_i = \mathcal{A}(\mathcal{D}_i)$
  - Average the model in predictions to get the $f = \frac{\sum_{i=1}^{m} f_i}{m}$
- This would reduce the variance to zero!
- Problem: we don't have $\mathcal{D}_1, \mathcal{D}_2, \cdots \sim P$

# Bagging ("Bootstrap aggregating")

- Algorithm invented by Leo Breiman in 1996
- As we don't have $\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_m \sim P$
- Idea: draw $\mathcal{D}_i$ uniformly **with replacement** from $\mathcal{D}$
  - $\mathcal{D}_i$ will have the same size as $\mathcal{D}$ but some of the members will be missing and some of them will be replicated
- Train on each of them $f_i = \mathcal{A}(\mathcal{D}_i)$
- Return the average $\hat{f}$
- The larger the $m$, the better, but at some point it will have diminishing results (but not a decrease in accuracy)

# Why is this working?

- We are not drawing i.i.d from the original distribution
- But it can be shown that the datasets are drawn from P, only not independently.
- In practice, it reduces variance very efficiently.

# Predicting uncertainty

- As the prediction is obtained as an average of $m$ classifiers, we can use these numbers to also obtain a variance
- This variance can be interpreted as the **uncertainty of the prediction**

# Predicting the test error

- Normally, it is difficult to predict the test error only from the training data.
- With bagging, we can provide an unbiased estimate for this.
- As $\mathcal{D}_i$ will have duplicated elements, there will be some elements of $\mathcal{D}$ which are not going to be in it (*out of the bag* elements)
  - For each training point $(\boldsymbol{x}_i, y_i)$
  - We can identify the datasets that do not contain it
  - Train a bagging model on these
  - Check the error on $(\boldsymbol{x}_i, y_i)$
  - Average over all $i$-s

# Random Forest

- A famous, and very well performing bagged algorithm
- Sample $m$ data sets $\mathcal{D}_1 \ldots \mathcal{D}_m$ from $\mathcal{D}$ with replacement
- For each $\mathcal{D}_j$ train a decision tree $f_j$
  - Modification: at each split randomly subsample $k \leq d$ features and only consider these at the split
- Final classifier: $f(x) = \frac{1}{m} \sum_{j=1}^{m} f_j(\boldsymbol{x})$

# Random Forest evaluation

- It is one of the **best-performing**, most popular and **easiest to use** classifiers / regressors

# What makes a machine learning algorithm easy/hard to use?

- Implementation difficulties? **NO**
  - Except in a class, you will probably not have to implement established algorithms by hand
  - Of course, if you propose a new algorithm, you will have to implement it.
  - There are many subtle tricks in implementations...
- What makes an algorithm hard to use is many sensitive hyperparameters
  - Works well if initialized with small random numbers, but diverges if the random numbers are larger than 0.02
  - Converges once out of 100 random initializations
  - Does not learn for learning rate $< 0.001$, oscillates if $> 0.0014$.
  - 7 hyperparameters, you need to get each of them just right

# What makes a machine learning algorithm easy/hard to use? (cont'd)

- Limitations and requirements on the type of features you can use
  - Dimensionality requirements
  - Specific type of encoding needed

# What makes Random Forest easy to use?

- Has only two hyperparameters: number of trees $m$ and split subset $k$
  - It is extremely unsensitive to both of them
  - You get them wrong, it will still mostly work
- A good choice for $k$ is the square root of the number of features $k = \sqrt{d}$
- $m$ - the higher the better.
  - in practice you stop when you run out of your time budget.
- Decision trees don't have complex requirements on features
  - Can take a mixture of discrete features, numerical features with different ranges etc.