1 General Concepts

- **Interpretability v.s. Prediction** Linear regression is very interpretable but has bad prediction abilities. Conversely, most statistical learning methods are less interpretable but better at prediction.
- **Bias-Variance Tradeoff** The expected MSE of a model can be written as

$$\mathbb{E}[(y_0 - \hat{f}(x_0)^2] = \underbrace{\text{Var}(\hat{f}(x_0))}_{\text{Variance of } \hat{f}} + \underbrace{[\text{Bias}(\hat{f}(x_0))]^2}_{\text{Bias of } \hat{f}} + \underbrace{\text{Var}(\varepsilon)}_{\text{Inherent Variance}}$$

  $\text{Var}(\varepsilon)$ is also called the irreducible error and lower bounds of the MSE of our model.

  A fundamental trade-off is between an overly capable and flexible model that can fit our data perfectly and an underly flexible model that cannot learn complicated underlying patterns. The former will have high variance because it fits random data samples more closely, whereas the latter will have high bias due to its inflexibility.
- **Bayes Classifer** The classifier with minimal expected MSE is the Bayes classifier:

$$\eta(x) = \arg\max_y P(Y = y|X = x)$$

  For a given observation x, the Bayes error rate is $1 - \max_y P(Y = y|X = x)$ and the overall Bayes error rate is

$$1 - \mathbb{E}_x[\max_y P(Y = y|X = x)]$$

  This is the minimum achievable error of any model but is rarely known in practice.
- **Overfitting** Overfitting occurs when the model is too capable and fits the random sample of acquired training data rather than true underlying dynamics. Effectively, overfitting arises when we fit the noise in the training data.
- **Evaluation and k-folds** We quantify overfitting by holding out a test set with which to evaluate. k-fold cross-validation makes this more robust by partitioning the dataset into k-folds. You then train k models, each one with a different set of $k-1$ folds and evaluate the trained model on the last held-out fold. By averaging over the k evaluations on held-out data, we get a more robust measure of our performance and better data efficiency. In the extreme case, setting k to the number of data points we have gives us the leave-one-out (LOO) error.

  However, if we pick our model based on its performance on an evaluation set, then we're indirectly optimizing the model for performance on the evaluation set. In practice, we should have a validation set (as above) and a separate test set which is used more sparingly. This lets us avoid fitting to the test set and get a better / less biased estimate of our performance.
- **Metrics** Consider the following table for N total examples (called the confusion matrix)

		Predicted	
		T	F
Actual	T	TP	FN
	F	FP	TN

  Then we can define the following metrics:
  - **Accuracy** $= (TP + TN)/N$. How often were you right?
  - **Sensitivity/Recall** $= TP/(TP + FN)$. The % of actually true events you correctly predicted/found. (True positive rate)
  - **Specificity** $= TN/(TN + FP)$. The % of actually false events you correctly predicted. (1 - false positive rate)
  - **Precision** $= TP/(TP + FP)$. The % of your of your predicted true events that are actually true.
  - **AUC** Area under the ROC curve (see below).
  - **F1** $= (2 \times TP)/(2 \times TP + FP + FN)$. Balance precision and recall.
- **ROC Curve** Plots how your sensitivity and specificity change if you change your threshold for what class you predict. To get it, sort all data points by increasing the order of the estimated probability of class 1. For each unique threshold (i.e., each unique value in your predictions), classify all points below the threshold as 0 and all points above the threshold as 1. Then, plot a point at $(x, y) = (FPR, TPR)$.
- **Micro-vs-macro Averaging** For classification problems with $C > 2$ classes we need a way to transform them to binary problems. For each class, treat it as a binary problem of correctly predicted or incorrectly predicted. In macro-averaging, you compute the metrics for each of your C classification problems and average the resulting metrics. In micro-averaging, you combine all your predictions of correctly predicted or incorrectly predicted, into one confusion matrix and compute the metric based on the pooled table. Macro-averaging will place more emphasis on rare classes as they are treated equally to common classes. Micro-averaging will place more emphasis on common classes as they outnumber rare classes.
- **Standardizing data** There are two common choices to make data easier to work with (putting all features onto the same scale).
  - True standardizing or normalizing. Set $x = \frac{x - \bar{x}}{\text{SD}(x)}$. I.e., subtract the mean and divide by the standard deviation for each feature.
  - Rescaling. Set $x = \frac{x - \min x}{\max x - \min x}$. I.e., rescale your data linearly to $[0, 1]$.
- **n-grams** Start first by consider the **bag-of-words** representation. For every word in our vocabulary, we create a variable x_i is 1 if word i is in a new piece of text and 0 if word i is not in the text. n-grams generalizes this by considering a sequence of n-consecutive words. E.g., $x_{i_1,i_2,\ldots,i_n}$ is 1 if word i_1 followed by words i_2 followed by ... up to i_n all appear. For instance "hello world" forms the bigram of hello followed by world. We also often remove stop-words that don't help like "the" and "a" and punctuation. n-grams help to recover positional information which bag-of-words representations cannot do. We can also consider parts of speech instead of specific words (e.g., noun, beginning or end of line, etc.). Bag-of-words and especially n-grams struggle with rare (combinations of) words, however, as the model will never see these particular n-grams during training.
- **TF-IDF** Bag-of-words and n-gram representations don't use information from repeated occurrences, which might be helpful. Let $f_{t,d}$ denote that term t appeared $f_{t,d}$ times in document d. We come up with term-frequency $\text{TF}_t = 1 + \log(f_{t,d})$ and $\text{TF} = 0$ when $f_{t,d} = 0$. We come up with the inverse document frequency $\text{IDF}_t = \log(N/N_t)$ where N is the number of documents and N_t is the number of documents containing term t. We now use the TF-IDF term as a feature during training $x_t = \text{TF-IDF}_t = \text{TF}_t \times \text{IDF}_t$.
- **Similarity** To measure the similarity of two inputs (particularly true for documents) we often use the cosine similarity: $\frac{\langle x_1, x_2 \rangle}{\|x_1\| \cdot \|x_2\|}$. When x_1 and x_2 point in the same direction (regardless of magnitude) $\text{sim}(x_1, x_2) \approx 1$ and are quite similar. When

x_1 and x_2 are orthogonal in direction $\text{sim}(x_1, x_2) \approx 0$ and are not at all similar. When x_1 and x_2 point in opposite directions $\text{sim}(x_1, x_2) \approx -1$ and are opposites of one another.

2 Simple Models

- **Logistic Regression** Usual linear regression doesn't work since probabilities should be bounded to $[0, 1]$ and linear models are unbounded. So, we instead make the linear regression model predict the log-odds (or logit) of a data point instead of its probability: $f(x) = \log \frac{p}{1-p}$. We can transform this to a probability by the sigmoid (or expit) function: $\sigma(x) = \frac{e^{f(x)}}{1-e^{f(x)}}$. Now the linear regression model $f(\vec{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$ has the following interpretation of its parameters:

  > A 1 unit increase to x_k increases the *odds* of being predicted as class 1 by a factor of e^{β_k}.

  The coefficients β are found by maximizing the likelihood

$$\begin{aligned}
&\arg\max_{\beta} \prod_{i=1}^{n} \Big(p(x_i;\beta)^{y_i} (1 - p(x_i;\beta))^{1-y_i} \Big) \\
&= \arg\max_{\beta} \sum_{i=1}^{n} \Big(y_i \log p(x_i;\beta) - (1-y_i) \log(1 - p(x_i;\beta)) \Big) \\
&= \arg\max_{\beta} \sum_{i=1}^{n} \Big(y_i \beta x_i - \log(1 + e^{\beta x_i}) \Big)
\end{aligned}$$

  To maximize, we take a derivative and set it to 0. But there's no closed-form solution, so instead, we use Newton-Raphson to find a zero (which requires using the second derivative). Logistic regression isn't very flexible since it's a linear model. It also doesn't have any tuning parameters to control its flexibility. But it's fast, interpretable, and doesn't overfit as easily. There's also an extension for multinomial regression.

- **Regularizing Linear Models** Least squares optimized linear models are the best linear unbiased models (BLUE), but trading off some bias in exchange for better variance might be worthwhile. Regularization introduces a penalty for complicated models making the new loss $L = \text{MSE} + \lambda \times \text{penalty}$ where λ is a hyperparameter. Two common penalties are based on penalizing the coefficients $\beta_0, \beta_1, \ldots, \beta_p$:

  - Lasso regression: penalty $= \|\vec{\beta}\|_1^1 = \sum_{i=1}^{p} |\beta_i|$. I.e., add an L^1 norm penalty on the coefficients. Penalizes large coefficients, but gives lee-way for very large coefficients.
  - Ridge regression: penalty $= \|\vec{\beta}\|_2^2 = \sum_{i=1}^{p} \beta_i^2$. I.e., add an L^2 norm penalty on the coefficients. Penalizes very large coefficients.
  - Scaled ridge regression: penalty $= \|\vec{\beta}\|_2^2 = \sum_{i=1}^{p} \beta_i^2 \,\text{Var}(x_i)$. This rescales the L^2 norm penalty since low variance features will need a relatively larger coefficient β to contribute to the overall prediction.

  While L^2 penalty usually gets better performance, L^1 penalty is good for feature selection. In particular, with an L^1 penalty useless features will have their coefficients shrunk to 0, whereas with an L^2 penalty useless features will just have a very small coefficient. You can also rewrite these under the dual formulation:

$$\min_{\beta}(\text{RSS} + \lambda \times \text{penalty}) \iff \min_{\beta} \text{RSS} \quad \text{where penalty} \leq \text{s}$$

  for each λ there is an s such that the two are equivalent. In this view we see that we restrict β to a ball of size s around the origin. Then, the closest point (according to L^2 norm or Euclidean distance since we're using MSE) to the ball is likely to be on a corner for an L^1 penalty but on the surface for an L^2 penalty. This is why lasso regression is good at feature selection, minima are usually on corners where a coefficient is set to 0.

- **k-Nearest Neighbours** The main idea is to estimate the Bayes classifier locally. k-NN is a memory-based algorithm that doesn't require training. Given a new point x, find the k closest points $x_1, \ldots, x_k$ in your dataset to k and predict by majority vote, picking the most common class amongst the $x_1, \ldots, x_k$. The closest point is usually picked by the Euclidean distance, but for binary features the intersection over union (Jaccard similarity) is another good choice. Another good one is cosine similarity: $\frac{\langle x_1, x_2 \rangle}{\|x_1\| \cdot \|x_2\|}$. Increasing k decreases the variance (since you have more samples voting) but also increases the bias (since the points that vote are further from new point). It's worth noting that k-NN is particularly sensitive to variable standardization/scaling. Break ties in majority vote by:

  - Random prediction amongst the most common classes.
  - By the prediction of the nearest neighbour (could be ill-defined if you have multiple nearest neighbours).
  - Keep increasing k until the tie is broken.

  Typically pick an odd k to minimize the number of ties. One benefit of k-NN is that it is really easy to find the LOO error. One variant of k-NN (albeit still rare) is to use a caliper instead, where you use all neighbours within a certain distance from you.

- **Naïve Bayes** Naïve Bayes arises from making the following simplifying assumption: given the class $y = y$, all x features are independent from one another. By making this assumption, we get that

$$\begin{aligned}
P(Y = y | X = x) &= \frac{P(X = x | Y = y) P(Y = y)}{\sum_{k=1}^{C} P(X = x | Y = k) P(Y = k)} \\
&= \frac{\pi_y \prod_{i=1}^{p} f_{y,i}(x_i)}{\sum_{k=1}^{C} \pi_k \prod_{i=1}^{p} f_{k,i}(x_i)} \\
&\propto \pi_y \prod_{i=1}^{p} f_{y,i}(x_i)
\end{aligned}$$

  Thus, we just predict the class that maximizes the numerator of Bayes rule. To predict the probabilities, we set the prior

$$\pi_k = \frac{|\{1 \leq i \leq n : y_i = k\}|}{n}$$

  and

$$f_{k,i}(x) = \frac{|\{1 \leq i \leq n : x_i = x, y_i = k\}| + L}{|\{1 \leq i \leq n : y_i = k\}| + d_i \times L}$$

  where d_i is the number of values x_i can take on and L is a parameter described below. If x is continuous, we instead estimate the probabilities from a marginal density, such as a Gaussian distribution (but this requires predicting $\mu_{k,i}$ and $\sigma_{k,i}$). Typically, however, Naïve Bayes works best for categorical data. One issue that arises is where a specific x-variable and y-variable has never been seen before, this sets the probability of the class to 0 (even if the other x-variables agree with it). We add Laplace smoothing to account for this: add L observations of each combination of x and y variables. Higher values of L shrink all distributions towards the mean. L is also a tuning parameter and can take on decimal values despite not being as intuitive. Naïve Bayes is very stable and fast because it breaks a multivariate problem into univariate problems.

3 Tree-based Models

- **Classification and Regression Trees (CART)** The idea is to sequentially partition our feature space into hyper-rectangles and fit a single constant to each hyper-rectangle. E.g., if $x_i < t$ predict a otherwise predict b, except a and b might also be decision trees further partitioning the space. For categorical variables, just split subsets of categories into each branch (often do one-vs-rest). The more splits you have the more interactions between variables are possible (this makes decision trees very powerful). Directly predicting all the rectangles that minimize a loss is infeasible, so we instead recursively split the space in half. For regression trees our criterion is usually MSE. For classification trees lots of criterions exist:
  - Classification error: $1 - \text{accuracy} = 1 - \max_k(\hat{p}_{m,k}$ where m is the leaf.
  - Gini index: $\sum_{k=1}^{C} \hat{p}_{m,k}(1 - \hat{p}_{m,k})$.
  - Entropy / information: $-\sum_{k=1}^{C} \hat{p}_{m,k} \log \hat{p}_{m,k}$

  The latter two criterions prefer pure leaves (fewer classes). Within a leaf we predict the constant that minimizes the loss of our criterion. Usually this is the average value in regression and the majority class in classification. Now to split a leaf (making it an internal node that partitions the space), we just pick the feature and split value (by brute force) that minimizes our criterion when we fit a constant to each leaf. Our stopping criterion varies, but there are common choices:
  - A maximum number of splits has been reached.
  - A maximum depth of the tree has been reached.
  - A minimum improvement threshold has been reached.
  - Each leaf has hit our minimum number of observations.

  After this, it's also common to prune the tree by removing splits one at a time to minimize the criterion

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} \ell(y_i, \hat{y}_m) + \alpha|T|$$

  where $|T|$ is the number of leaves and ℓ is our loss. Increasing α means smaller trees with more bias but less variance. Trees are especially good at dealing with missing values, here's a few ways:
  - Add a missing value category. This often means just assigning missing values to one of the two branches
  - Just pick a child to go to (random or largest).
  - Create a surrogate tree where you pick the branch you'll go down split based on other, non-missing values. Create a split (or multiple) based on the other variables that picks which of the two branches you go down by minimizing the loss using those fixed predictions / branches.

- **Bagging** Bagging (or bootstrap aggregating) can in theory be applied to any algorithm, but is particularly popular for tree methods. The idea is simple: draw B bootstrap samples (samples of size n with replacement) from your data and learn a model on each bootstrap sample. Then, just predict the average value (regression) or majority (classification). Even better, have each model predict a probability of class k and predict the average probability. Bagging removes interpretability, but leads to random forests.

- **Random Forest** As in bagging, build m trees from m different bootstrap samples. Within each tree, each time you make a split pick a subset of the p features and only choose a split from among those variables (commonly use $\sqrt{p}$ of all variables). This helps to de-correlate the tree and better use all variables. Often specify maximum number of splits or minimum node size to determine when to stop. Often keep adding trees until the test error stops improving. One benefit of random forest is that you can use out-of-bag data points for testing. For each datapoint, get the predictions of all trees that don't use it in training and average their predictions. Using the error from these trees gives a good estimate of the error. Variable importance is often also based on the out-of-bag samples.

- **Variable Importance** We can estimate how important an x variable is to the overall model by destroying its predictive power and finding the testing accuracy. Do this by shuffling/permuting all the features of the variable and having the model predict with the permuted data. The decrease in prediction accuracy is proportional to the importance of the variable. By repeating this permutation process we can get a more stable estimate of importance.

- **Adaboost** The earliest form of boosting, uses lots of weak learners to form a strong learner. Start with a uniform distribution/weights over your data. For $m = 1, \ldots, M$ fit a classifier G_m to your (weighted) data and compute the (weighted) error. Set the weight of classifier $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$ and set the weight of each datapoint to $w_i = w_i \cdot \exp(\alpha_m \mathbb{1}[y_i \neq G_m(x_i)])$. Intuitively, the next model is fit to the data we greater emphasis on datapoints that the previous model incorrectly classified. The overall classifier is

$$G(x) = \text{sign}(\sum_{m=1}^{M} \alpha_m G_m(x))$$

  One can prove that Adaboost's error will go to 0 as $M \to \infty$ (see CS 485). The weights w_m should be interpreted as a distribution over the data which gets renormalized to sum back to 1.

- **Boosting** Now we consider general boosting, which is a generalization built off of Adaboost. The overall classifier will instead be

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m)$$

  where our b's are basis functions / weak learners. Boosting is a sequential, additive model where after learning $b_1, \ldots, b_m$ we hold these constant and fit b_{m+1} to the residuals of b_m. In particular, we learn b so as to minimize the loss

$$\frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_{m-1}(x_i) + \beta_m b(x_i; \gamma_m))$$

  and update the function $f_m = f_{m-1} + \beta_m b_m(\cdot\,; \gamma_m)$.

  Think of the loss as a function of the target and our predictor $\ell(y, f)$. We want to minimize the overall loss w.r.t. our function $\arg\min_f L(f)$. We can minimize this loss by setting the derivative to 0, but that means finding an f such that $L'(f) = 0$. Since we're trying to make the derivative of the loss 0, let's fit our next basis function b_m to the negative derivative of the loss of the last model (this is a step in the direction of minimum loss). This is the pseudo residual of f_{m-1}: $r_m = \frac{\partial L(f)}{\partial f}\Big|_{f=f_{m-1}}$. Now our targets are the negative residuals and our fitted values are the values that will set the derivative (residuals) to 0. We can apply Newton-Raphson to see how we should get to an f_m that gets a derivative of 0

$$f_m = f_{m-1} - \underbrace{\frac{\ell'(y_i, f_{m-1})}{\ell''(y_i, f_{m-1})}}_{=\gamma_{j,m}}$$

  where $x_i \in R_{j,m}$. One can think of this equation as minimizing $\sum_{i=1}^{n} \ell(y_i, f_{m-1}(x_i) + \gamma_{j,m})$ w.r.t. $\gamma_{j,m}$. So to summarize the (MART) algorithm:

Initialize f_0
for $m = 1, \dots, M$ **do**
  Compute pseudo-residuals: $r_{i,m} = -\dfrac{\partial \ell(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)}$ for $i = 1, \dots, n$
  Fit a new regression tree to the psuedo residuals
  Fitted values of the leaves of the tree are

$$\gamma_{j,m} = -\sum_{i:i\in R_{j,m}} \frac{\ell'(y_i, f_{m-1}(x_i))}{\ell''(y_i, f_{m-1}(x_i))}$$

  Update $f_m(x_i) = f_{m-1}(x_i) + \gamma_{j,m}$ where $x_i \in R_{j,m}$
return f_M

- **Tuning Boosting Models** We usually specify a fixed number of leaves per tree or a fixed depth of each tree. We also specify the number of trees/iterations M. We can additionally introduce a learning rate / shrinking parameter ν by multiplying $\gamma_{j,m}$ by ν, making the steps taken by our iterations smaller. It's also common to fit each tree in boosting with a different random subset (e.g., 50%) of observations. This is not a bootstrap sample and is taken without replacement. Similar to random forests we also often choose a random subset of the available features for each split.
- **XGBoost** XGBoost improves over usual (MART) boosting by making engineering improvements adding a regularization term. We now regularize the loss we minimize w.r.t. to $\gamma_{j,m}$ by

$$\sum_{i:i\in R_{j,m}} \ell(y_i, f_{m-1}(x_i) + \gamma_{j,m}) + \frac{\nu}{2}\sum_{j=1}^{|T|}\gamma_{j,m}^2 + \alpha|T|$$

We now have an L^2 penalty on our predictions $\gamma_{j,m}$ and a penalty on the size of our trees $\alpha|T|$. We also simplify the first term (our usual error) by a Taylor expansion around $a = f_{m-1}(x_i)$ for each $i = 1, \dots, n$.:

$$\ell(y_i, f_{m-1}(x_i) + \gamma_{j,m}) \approx \ell(y, f_{m-1}(x_i)) + g_i\gamma_{j,m} + \tfrac{1}{2}h_i\gamma_{j,m}^2$$

where $g_i = \left.\frac{\partial \ell(y_i,f)}{\partial f}\right|_{f=f_{m-1}(x_i)}$ and $h_i = \left.\frac{\partial^2 \ell(y_i,f)}{\partial f^2}\right|_{f=f_{m-1}(x_i)}$. Written using this Taylor expansion, we get an overall loss of

$$\sum_{i=1}^{n}\ell_i + \sum_{j=1}^{|T|}\underbrace{\left[\gamma_{j,m}\sum_{i:i\in R_{j,m}} g_i + \frac{\gamma_{j,m}^2}{2}\left(\nu + \sum_{i:i\in R_{j,m}} h_i\right)\right]}_{-\text{similarity}} + \alpha|T|$$

where $\ell_i = \ell(y_i, f_{m-1}(x_i)$ is the loss of our previous iteration. Solving for $\gamma_{j,m}$ we now get our fitted values being

$$\gamma_{j,m} = -\frac{\sum_{i:i\in R_{j,m}} g_i}{\nu + \sum_{i:i\in R_{j,m}} h_i}$$

For the Gaussian object (MSE) this simplifies to

$$\gamma_{j,m} = -\frac{\text{RSS}}{\nu + \#\text{ of residuals}}$$

To efficiently compute this more complicated loss in each split, we only evaluate the terms which change, the terms in the square brackets in our loss, called the negative similarity. Plugging our optimal γ into the formula for similarity, we get

$$\text{similarity} = \frac{1}{2}\cdot\frac{\left(\sum_{i:i\in R_{j,m}} g_i\right)^2}{\nu + \sum_{i:i\in R_{j,m}} h_i}$$

We then just pick the split which maximizes our gain:

$$\text{gain} = \text{similarity}_{\text{left}} + \text{similarity}_{\text{right}} - \text{similarity}_{\text{parent}} - \alpha$$

where the α comes from our penalty on tree size. But α doesn't affect picking which split has the highest gain. In practice we just grow trees until some stopping criteria is reached (e.g., max depth) and prune leaves with less than α gain.

Finally, XGBoost also has engineering improvements. Most important of which is splitting continuous values only at set quantiles (not every possible split). XGBoost also naturally handles missing values by sending missing values to the branch which maximizes the gain of the split. It also uses subsampling of datapoints and only splitting on a subset of features as in normal boosting. Finally, parallel computing and careufl memory management help make it more efficient.

- **Variable Influence** The influence of variable x_i in a tree is the sum of redudction in the residual sum of squares across all splits on variable x_i. For a method with multiple trees, we then usually average this over all trees:

$$I_i = \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{|S(T_m)|}(\text{RSS}_j - \text{RSS}_{j,\text{left}} - \text{RSS}_{j,\text{right}})\mathbb{1}[\text{split } j \text{ is on } x_i]$$

where $S(T_m)$ is all the internal/splitting nodes in tree T_m.

- **Recycled Predictions** We could also determine how changing the value of a variable changes predictions. For each datapoint, set x_j to a fixed value but keep all other variables as usual. Then plot the average prediction over all data points for each fixed value of x_j to get an idea of the overall impact of x_j.

4 Complex Models and Ensembling

- **Support Vector Machines** SVMs arise from the idea that for linearly separable data multiple lines may separate our data perfectly. SVMs will pick the line which maximizes the margin, distance from the decision boundary to the closest observation. The closest observations (of which there must always be at least one of each class) are called support vectors. Encode response at $y \in \{-1, 1\}$. SVMs are always halfspaces of the form $\hat{y} = \text{sign}(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)$. We want to maximize the margin M where $y(\beta_0 + \langle\beta, x\rangle) \geq M$ (which is always > 0 when we're correct). The margin can also be written as $\frac{y(\beta_0+\langle\beta,x\rangle)}{\|\beta\|_2}$. Since we only care about the sign, we can fix the numerator (or the denominator) arbitrarily. Let's set the numerator to be 1, then we want to maximize $\frac{1}{\|\beta\|_2}$. This is the same as $\min_\beta \frac{1}{2}\|\beta\|_2^2$ subject to the constraint $y(\beta_0 + \langle\beta, x\rangle) \geq 1$. Using Lagrangian duals, we get

$$\min_\beta \max_{\alpha\geq 0} \tfrac{1}{2}\|\beta\|_2^2 - \sum_{i=1}^{n}\alpha_i(y_i(\beta_0 + \langle\beta, x_i\rangle - 1)$$

But we often have non-linearly separable data, so we allow some slack: $y_i(\beta_0 + \langle\beta, x_i\rangle) \geq M(1 - \varepsilon_i)$ where $\sum_{i=1}^n \varepsilon_i \leq C$. Equivalently, we just limit the maximization to $\max_{0\leq\alpha\leq C}$. Note our prediction (1) correct when $\alpha_i = 0$ (2) incorrect when $\alpha_i = C$ and (3) correct and a support vector when $0 < \alpha_i < C$.

Finding the minim w.r.t β we get $\beta_i = \sum_{i=1}^n \alpha_i y_i x_i$. But plugging this into a new evaluation point we find that this is the same as

$$\hat{y} = \text{sign}\left(\beta_0 + \sum_{i=1}^{n}\alpha_i y_i\langle x_i, x\rangle\right)$$

Thus far we're limited to halfspaces / linear classifiers. What if we lift our data to a higher dimensional space, e.g., by making

new features $\phi(x) \mapsto (1, x_1, \ldots, x_p, x_1^2, x_1x_2, x_1x_3, \ldots, x_p^2)$? This would be a lot slower for our computation because we now have $1 + p + p^2$ features... But the inner product is fast, it's just $\langle \phi(x), \phi(z) \rangle = (\langle x, z \rangle + 1)^p$! This is called a reproducing kernel, where ϕ has a special form to make $\langle \phi(x), \phi(z) \rangle$ fast. The polynomial kernel $k(x, z) = (\langle x, z \rangle + 1)^d$ is one example, but a better choice is often the Gaussian kernel (or radial basis function):

$$k(x, z) = \exp\left(-\gamma \sum_{i=1}^{p} (x_i - z_i)^2\right)$$

Other choices exist, like the sigmoid kernel, Laplace kernel, etc. Note that SVMs are incredibly sensitive to scales. Continuous variables should be standardized by subtracting the mean and diving by the standard deviation. Note also that SVMs produce a sort of score $\text{score}(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i y_i k(x, x_i)$. If you want probabilities, just train logistic regression on the score and recover the probability from that.

- **Neural Networks** Given input x, a single stage's forward pass puts x through a linear transformation $z = Wx$ followed by a non-linearity $\text{out} = \sigma(z)$. σ is called the activation function, common choices include ReLU, GeLU, Sigmoid, tanh, etc. Stacking multiple layers as such gives you a neural network (MLP to be specific). We update the weights W by a stochastic gradient descent (with learning rate η). To do this, we need to find the derivative of the loss w.r.t. the weight W by backpropagation (repeated application of the chain rule).

  Considerations: When the chain of partial derivative is long, a lot of the partial derivative will either be < 1 and cause vanishing gradients or > 1 and cause exploding gradients. This is part of why ReLU is preferred and is leads to the success of ResNets. To stabilize the variance of neural networks, weights should be initialized with Xavier initialization. Let fan_{in} denote the number of input variables to a particular node. Then all weights associated with that node should be initialized according to a $N(0, \frac{1}{\text{fan}_{in}})$ distribution. This will make all nodes have outputs with variance of 1 (in expectation). To regularize neural networks, common techniques include weight-decay by adding a penalty $\lambda \sum_{i=1}^{N_W} w_i^2$ (or absolute value for L^1 penalty) to encourage smaller weights, as in ridge regression. Another form of regularization is dropout where during training you randomly remove edges with a certain probability to encourage learning redundancies. It's also better to standardize inputs for neural networks. Neural networks can struggle to converge and are prone to overfitting, particularly for tabular data and in smaller datasets.

- **Ensembling** We can improve our accuracy and variance by training multiple models (akin to bagging). Train each model on a subset of the data and predict the average (regression) or majority (classification) answer. Often ensembling is done with models of the same type.

- **Stacking** A slightly more sophisticated version of ensembling is stacking. It produces predictions by training a logistic regression model whose inputs are the outputs of the base models. For each model, use k-fold cross-validation and training on $k-1$ folds and predicting on the held out fold. Then combine all the predictions on held-out folds to train a logistic regression model. Finally, retrain the base models using all the data. The final stacked model runs each base model and passes all the outputs as the input to the logistic regression model.

  Often stacking uses base models of different types. One variant is to also pass through the input x-variables as an input to the logistic regression model. Another variant is to use "extremely randomized trees" as one of the base models. These are a variation of random forests where the split point for each feature is drawn at random and then the best variable (with the random split) is chosen. When also using a subset of features as usual, these have very high bias but lower variance. These can sometimes passthrough a bit more information due to their poor prediction ability.

  But should we always weight each model the same for all datapoints? Some models might do better in certain cases than others. If we have some small number of relevant meta features (e.g., number of reviews), we can come up with a weight function and have our logistic regression model be of the form

$$\log \frac{p(x)}{1 - p(x)} = \beta_0 + \sum_{i=1}^{m} \sum_{j=1}^{k} \beta_{i,j} f_j(x) g_i(x)$$

  where $f_j(x)$ is one of the k weights based on meta-features and we have m base models $g_1, \ldots, g_m$. However, this is more complicated and requires more hands-on engineering, but it can be effective.