ML from a statistical perspective uses probability models to simulate data however the probability models don't work well in simulating the data in real life.

Computers should learn from data recognize pattern and suggest what is best without a lot of hard coding

ML can perform better than if-else statements written by humans and the probability models. However, the reason behind prediction of complex models is sometimes unknown

ML has 3 main categories

1. Supervised learning - algorithm studies labeled data to know what label to give to unlabelled data. Categories of supervised learning
    - Classification
        - predict category (state) that the data belong to
        - there can be as many categories considered during training
        - The task is to find decision boundary that separate data of different categories
    - Regression - predict numerical values for data

    Applications : spam filtering, recommendation system

2. Unsupervised learning - no labels are given to algorithms. Applications : grouping data, reducing no. of features, recommendation system

3. Reinforcement learning - train algorithms to learn from taking actions. Give rewards to algorithms on correct actions. Applications : self-driving cars, game playing agent

- Supervised is used in business more than unsupervised learning
- Reinforcement learning is hard to train and least used in business

2 main families of predictive ml algos are classification algo and regression algo

Deep learning (DL)

- better prediction ability (sometimes) than ML algorithms
- Can be used for all 3 categories of ML
- Hard to understand how or why model make certain predictions (complex and flexible)
- Requires a lot of data and computing power

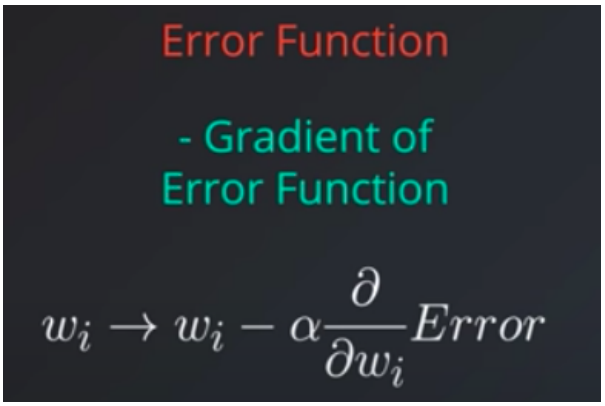
Data with errors and biases can lead to models with errors and biases. Should check if model has bias (real world validation) before deploying

Linear regression algo (regression)

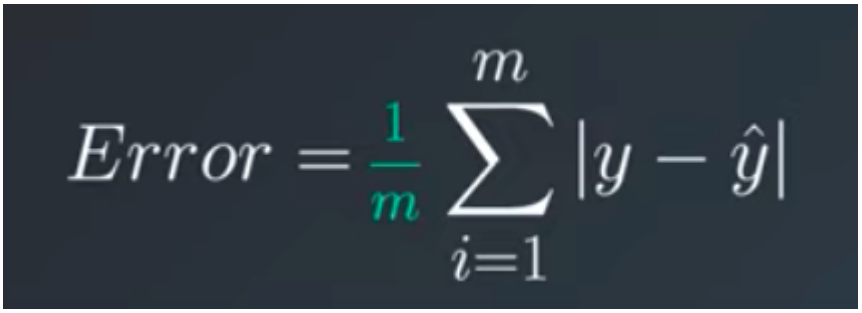
- Finds best fitting line to data
  $y = wx + b$
- Start with random w and b, adjust weights using one of
    - Absolute trick

1. sign = sign(y - ypred)
2. w = w + sign * lr * x
   b = b + sign * lr

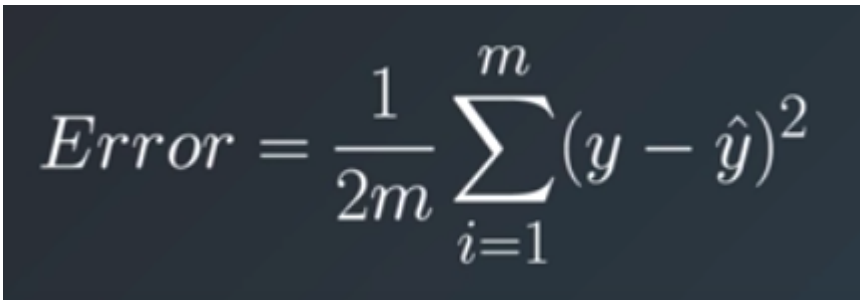
- Square trick - adjust weight based on error, larger error result in larger change
  1. error = y - ypred
  2. w = w + error * lr * x
     b = b + error * lr



- Gradient descent - minimise error by going into negative direction of error function gradient
  - Error functions for linear regression
    - Error functions should not make errors cancel out
    - MAE = mean(abs(y - ypred))



    - MSE = 0.5 * mean((y - ypred) ** 2)



    - ½ is for ease of taking derivative

- y - ypred != perpendicular distance from point to regression line
  != actual distance from point to regression line
- y - ypred = **vertical** distance from point to regression line

- Gradient descent with MAE = absolute trick
- Gradient descent with MSE = square trick

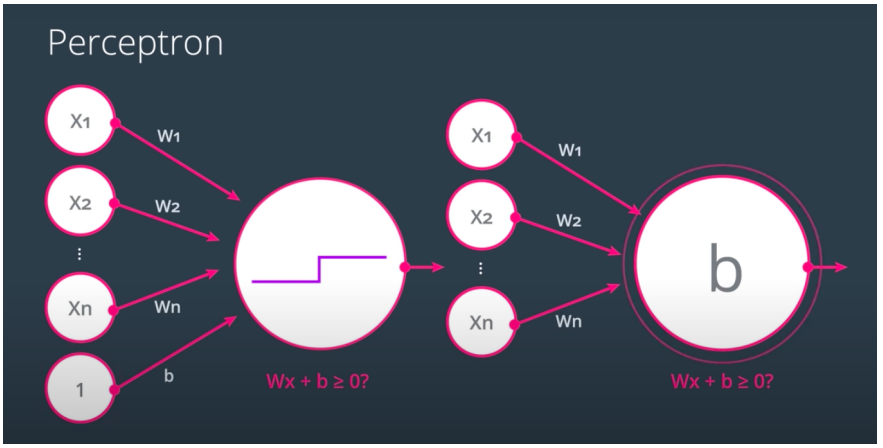
- Batch gradient descent - use all data points per weight update and repeat for many iterations (**slow**)
- Stochastic gradient descent - use one data point per weight update, repeat for all data points and repeat for many iterations (**slow**)
- Mini batch gradient descent - split data into many small batches, use one mini batch per weight update, repeat for all mini batches and repeat for many iterations

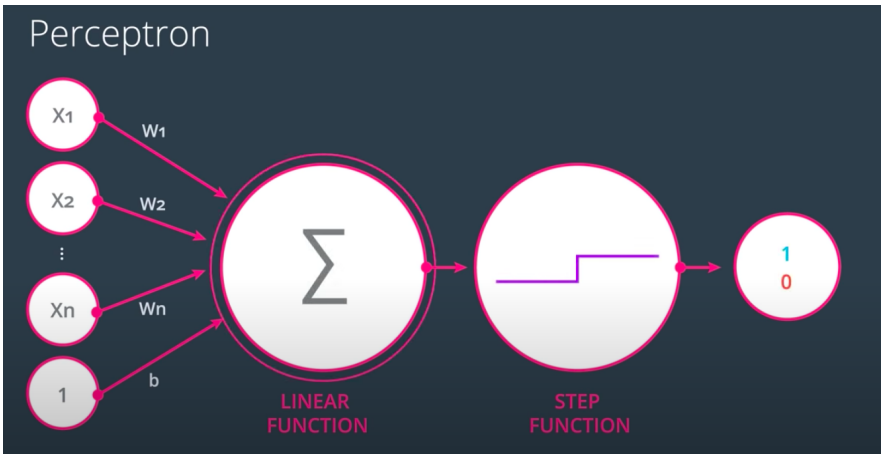
- Predictor = independent variable = feature
    - Predictor is a variable you're looking at in order to make predictions about other variable
- Target = dependent variable = predicted variable
- Multiple Regression = regression with more than 1 independent variable
- Adding relevant predictors can help get better prediction
- One predictor linear regression gives a line
- A multiple linear regression gives a hyperplane
- Adding more predictor variables, linear regression results becomes harder to visualise but way to find line / hyperplane is the same
- Can solve linear regression in one step using linear algebra to find optimal weights minimising MSE
- Linear algebra method solve equation with n (# parameters) unknowns but for large n linear algebra can be time consuming and computationally expensive
- Linear regression assumptions
    1. Linear data i.e. linear relationship between features and target
    2. Linear regression is sensitive to outliers i.e. outliers might pull model towards them
- Polynomial regression - for non-linear data i.e. when features don't have linear relationship with target, create more features to make linear regression easier to fit data. Model is a polynomial curve.
- Regularisation - to make algorithm pick simpler models, we can charge more penalty as the model get more complex (take complexity into account when computing error)
- L1 regularisation - loss = loss + lambda * sum(abs(w))
    - Computationally inefficient to take derivative of absolute function if weights are non-zero
    - Make weights zero, automatic feature selection
- L2 regularisation - loss = loss + lambda * sum(w**2)
    - Computationally efficient to take derivative
    - Produce small non-zero weights, no automatic feature selection
    - L2 charges higher penalty compared to L1 due to squaring
- Low lambda (no regularisation) give overfitting model
- High lambda (too much regularisation) give underfitting model
- Model that require low error is ok to be complex
- Model that should run fast need to be simple
- Feature scaling = a way of transforming data into a common range of values, 2 common scalings are
    1. Standardising:
        - (feature - mean(feature)) / sd(feature)
        - standardised value can be interpreted as the number of standard deviations the original height was from the mean.

2. Normalising
    - (feature - min(feature)) / (max(feature) - min(feature))
    - give normalised features with value between 0 and 1

- When to use feature scaling
    1. When the algorithm uses a distance-based metric to predict
        - not scaling data may lead to drastically different (and likely misleading) predictions
    2. When regularisation is used
        - Having small and large range features, regularisation is going to unfairly punish the feature with the small range
        - Features with small ranges need to have larger coefficients compared to features with large ranges in order to have the same effect on the outcome.
        - Regularisation would rather remove the feature with the large coefficient, since that would reduce the regularisation term the most.
- feature scaling can speed up convergence of machine learning algorithms

Perceptron algo (classification)
- is building block of neural networks
- 2 ways to represent perceptron





- Use linear decision boundary (wx + b) to separated data by target classes
    - if wx + b >= 0 predict positive class i.e. class 1

    - if wx + b < 0 predict negative class i.e. class 0

- Perceptron apply step function to continuous output from wx + b
- Decision boundary is # feature - 1 dimensional hyperplane
- No need to separate training data perfectly, separate most of the training points correctly is enough

- Perceptrons can be used as logical operators
    - And operator wx + b >= 0 when x is vector of pure ones
    - To go from and operator to or operator we could decrease bias magnitude or increase weights
    - Not operator takes only one input (other inputs are ignored) and return opposite of the input

- Perceptron trick
    1. Start with a random line
    2. Classify points using line in 1., check which points are incorrectly classified
    3. For each incorrectly classified points
       3.1 Take the derivative of the perceptron line equation w.r.t. each weight
       3.2 Modify weights
           w = w + sign(y - ypred) * lr * derivative in 3.1 for the weight
       ** y and ypred are binary
    4. Perceptron error = abs(wx + b)

- Perceptron trick will make the decision boundary come closer to incorrectly classified point until the line passes the point making it correctly classified

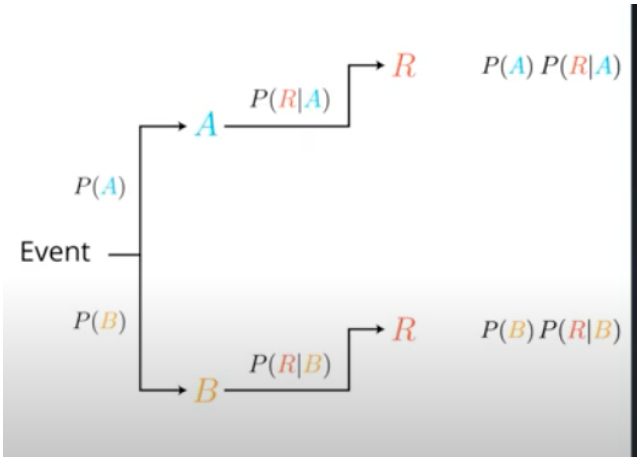
Decision tree algo (regression and classification)
- Decision tree find conditions that separate data best
- Numeric features are used to create branching condition using threshold
- The more homogeneous set of objects is, the less its entropy is
- The more entropy of a set, the less knowledge about the set we have e.g.
    - Box of pure red balls give us high knowledge that we will definitely get red ball when picking a ball from the box (low entropy)
    - Box of half red, half blue balls give use low knowledge (high uncertainty) that a ball we pick will be red (high entropy)
- For probability, sums are better than product as result from product of probabilities can be very small number

- Log is a function to turn products into sums
- Decision tree uses log base 2
- Entropy = - sum(p * log2(p)) where p is probability or proportion of each category
- The minimum value of entropy is 0 (all elements are the same)
- The maximum value is achieved when the probability of all categories are the same, (upper limit increases with the number of different categories)

- Information gain = reduction in entropy = parent entropy - weighted sum of children entropy
- Decision tree chooses a feature which given largest information gain as split criterion
- Decision tree hyperparameters
    - Maximum Depth
        - the largest edge count between the root and a leaf
        - A tree of maximum depth k can have at most 2** k leaves
    - Minimum number of samples to split (min_samples_split)
        - if a node has fewer samples than min_samples_split, it will not be split
        - min_samples_split doesn't is not minimum size of leaves
        - Large min_samples_split may result in the tree not having enough flexibility to get built, and may result in underfitting
    - Minimum number of samples per leaf (min_samples_leaf)
        - Helpful in the split problem of having many samples in one child, and very few (e.g. 1) on the other (would be a waste of resources and time)
        - specified as a float; for example, 0.1, or 10%, a particular split will not be allowed if one of the leaves that results contains less than 10% of the samples in the training dataset

Naive Bayes algo (classification)

- Probabilistic algo that computes the conditional probabilities of the input features and assigns the probability distributions to each of possible classes
- Beneficial because easy to implement, fast to train
- Example application : NLP
- can be trained to classify data into multi-class categories
- Prior probability = probability before knowing new information
- Posterior probability = probability after information arrived
- Bayes theorem switches from what we know to what we infer (what we want to know)
- What we know is P(A), P(R | A), what we infer is P(A | R) probability of A after R has occurred



$$P(A|R) = \frac{P(A)P(R|A)}{P(A)P(R|A) + P(B)P(R|B)}$$

$$P(B|R) = \frac{P(B)P(R|B)}{P(A)P(R|A) + P(B)P(R|B)}$$

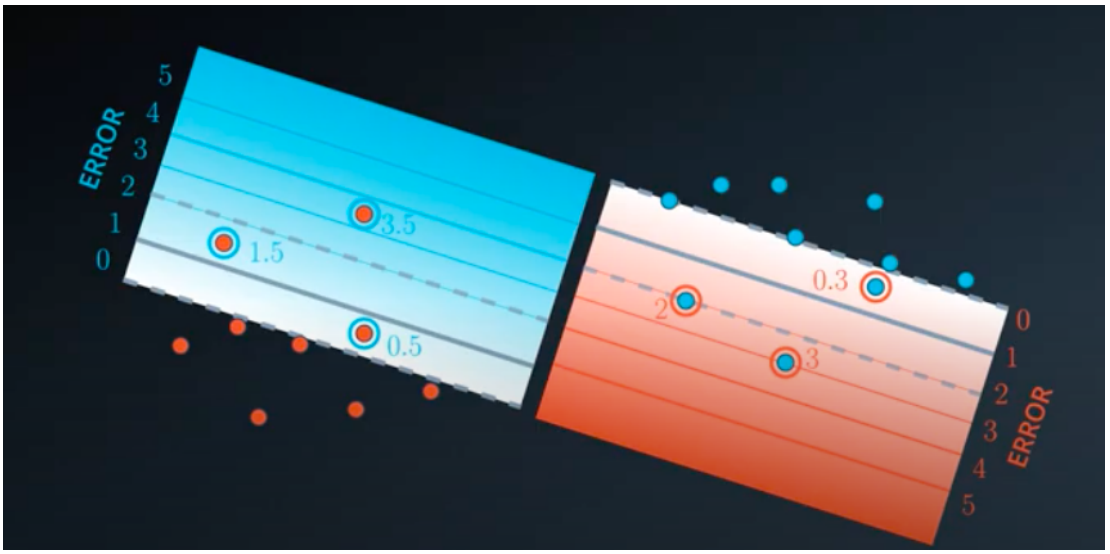
- P(A) and P(B) are prior probabilities (what we know before event R occurred)
- P(A | R) and P(B | R) are posterior probabilities (what we inferred after R has occurred)
- For NLP, Naive Bayes take occurrence of word into account but not the order of word
- Why Naive Bayes is naive

    - It assumes that events are independent
        - P(A & B) = P(A) * P(B) when A and B are independent events
        - P(A | B) is proportional to P(B | A) * P(A)
        - P(A | B & C) = P(B & C | A) * P(A) = P(B | A) * P (C | A) * P(A)

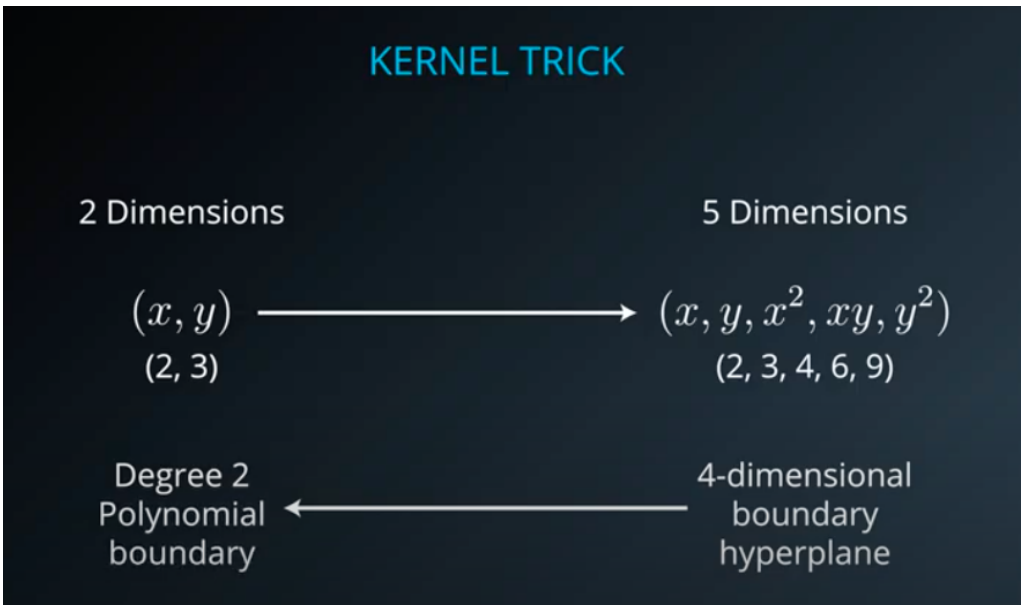
- Sensitivity = True Positive Rate = TP / (TP + FN)
- Specificity = True Negative Rate = TN / (TN + FP)

SVM

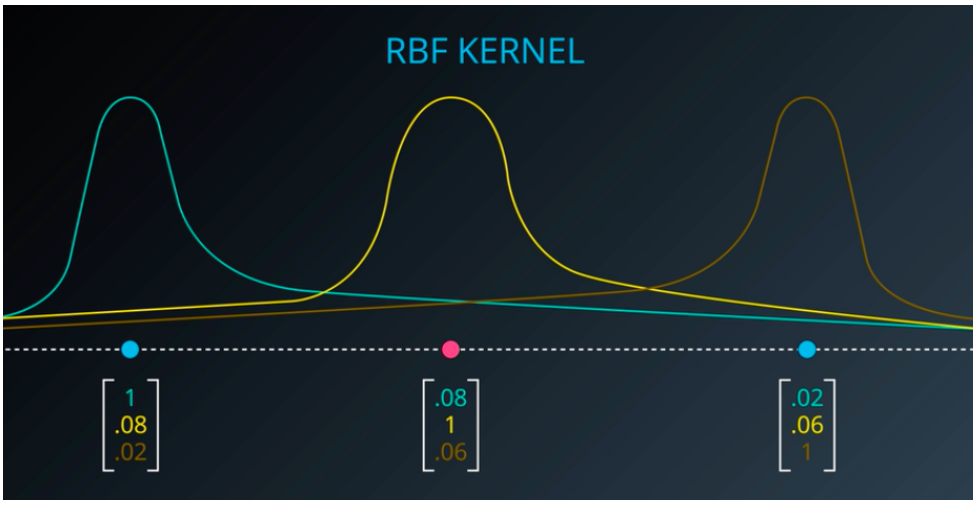
- Try to find decision boundary (for classification) that maintains largest distance from data points
- SVM decision boundary is surrounded by 2 parallel lines. The parallel lines are equally far from main decision boundary
- The goal is to maximize the distance (margin) between the parallel lines
- In SVM, misclassified points and points that lie in margin (margin errors) are used to penalize model
- The goal for svm is to minimize classification error and margin error
- Classification error start to increase as prediction exceed margin of current class
- SVM classification error
    - Error positive class = abs(wx + b - 1)
    - Error negative class = abs(wx + b - (-1)) = abs(wx + b + 1)

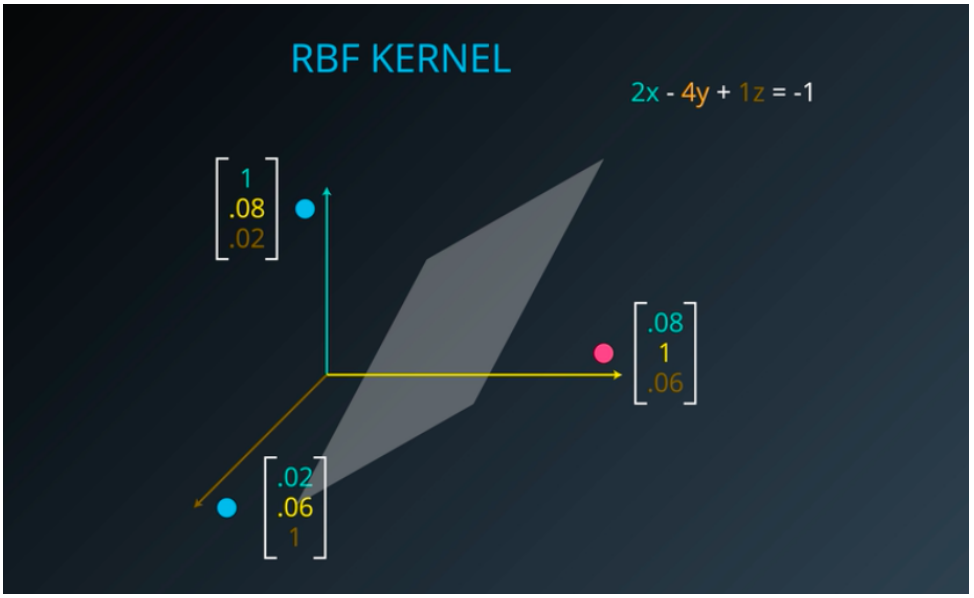


- margin = 2 / || w || = distance between parallel lines that surround model
- margin error = || w || ** 2 (similar to L2 regularization)
    - Large margin give small error, small margin give large error
- SVM total error = C * classification error + margin error
    - Large C means focusing on classifying (training) data points correctly but may give decision boundary with small margin
    - Small C means focusing on larger margin by allowing misclassifications
- Kernel SVM use kernel function to transform features into higher dimensional space to find decision boundary in the high dimensional space, then the decision boundary found is mapped back to original the space

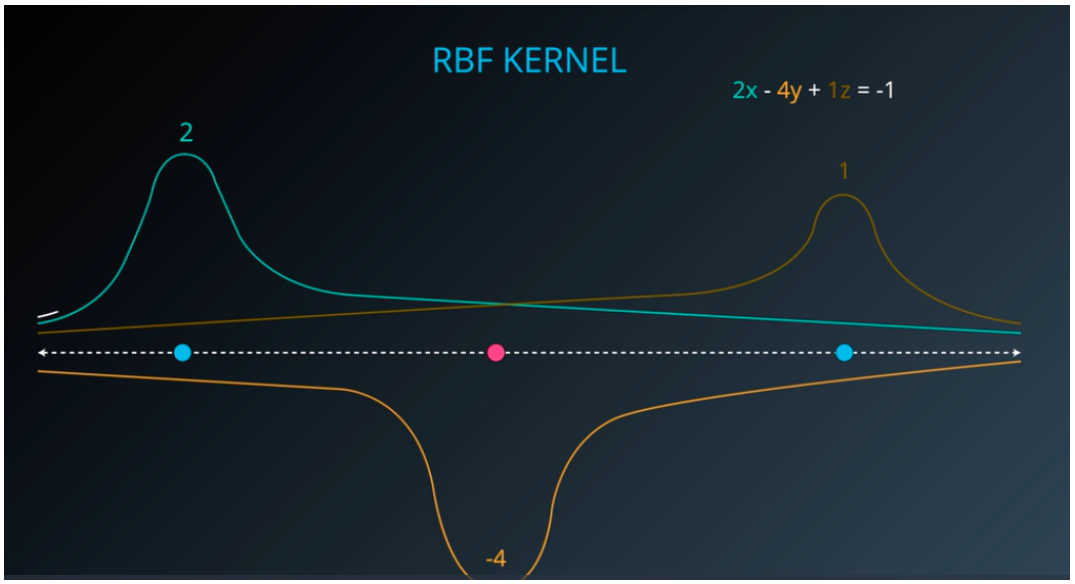


where x and y are features of a training example

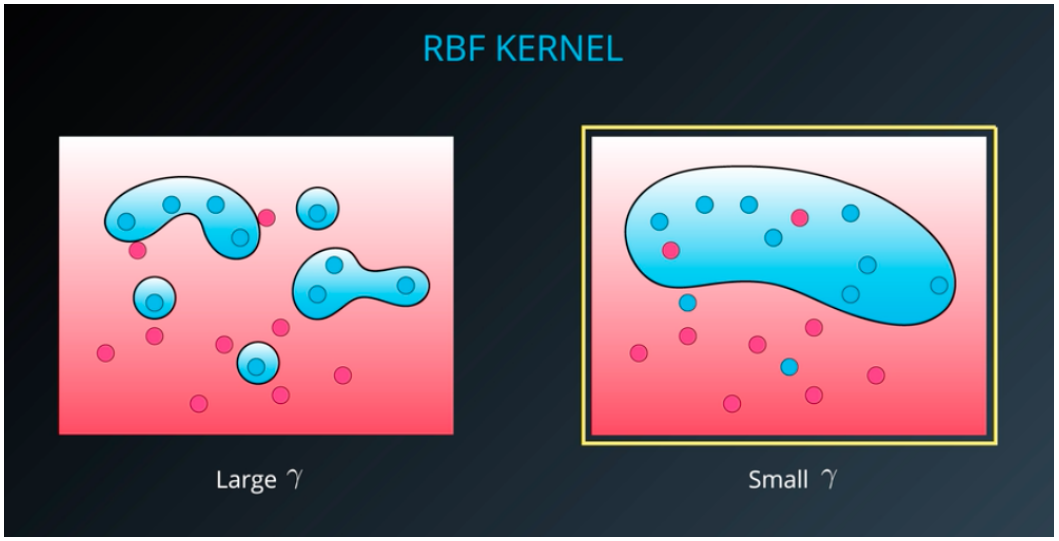




- RBF kernel SVM compute similarity of examples with each training example then use the similarity score coordinates and target class to find decision boundary
- Think of SVM with RBF kernel like fitting a gaussian graph to each training point. The gaussian graph is centered at the training point the gaussian belongs to



  half = (max_coeff - min_coeff) / 2= (2 - (-4)) / 2 = 6 / 2 = 3
  middle = max - half = 2 - 3 = -1
- The gaussian graphs can be wide or skinny depending on gamma parameter
  - large gamma give skinny graph
  - small gamma give wide graph
- gamma = 1 / (2 * (sigma ** 2)) where sigma is width of gaussian curve



3 different ways that SVMs can be implemented

1. Maximum Margin Classifier - when data can be completely separated, the linear version of SVMs attempts to maximize the distance from the linear boundary to the closest points (called the support vectors).
2. Classification with Inseparable Classes - data in the real world is rarely completely separable, a hyper-parameter called C (ranges between 0 and infinity) determines how flexible we are willing to be with the points that fall on the wrong side of the decision

boundary. When C is large, you are forcing your boundary to have fewer errors than when it is a small value.

Note: setting too large value of C for a particular dataset, might not make SVM convergence because data cannot be separated with the small number of errors allotted with such a large value of C

3. Kernels in SVMs allow us to separate data when the boundary between them is nonlinear
    - By far the most popular kernel is the rbf kernel which allows you to classify points that seem hard to separate in any space
        - The RBF kernel uses a density based approach that looks at the closeness of points to one another.
        - RBF kernel hyper-parameter gamma
            - a large gamma is similar to having a large value of C (algorithm will attempt to classify every training point correctly)
            - a small gamma will try to cluster in a more general way that will make more mistakes, but may perform better when it sees new data

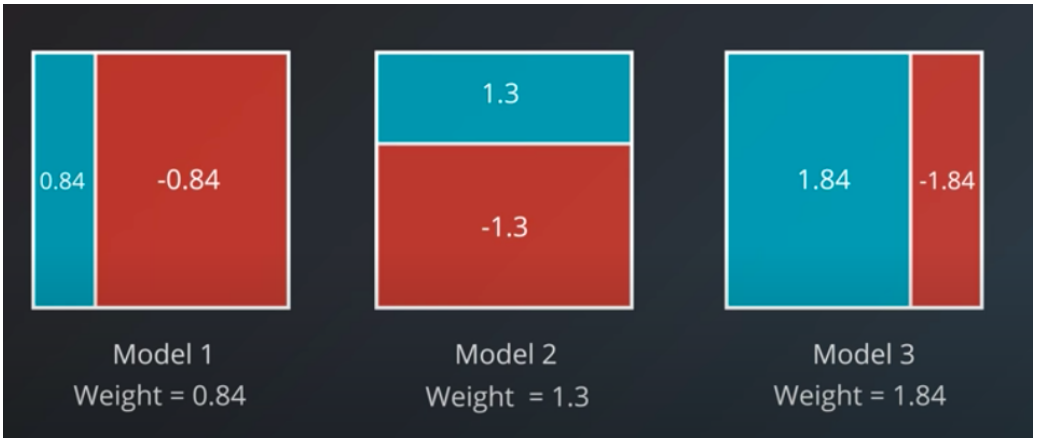
Bias: When a model has high bias, it doesn't do a good job of bending to the data.
- An example of an algorithm that usually has high bias is linear regression.
- Even with completely different datasets, we end up with the same line fit to the data.

Variance: When a model has high variance, this means that it changes drastically to meet the needs of every point in our dataset or it is extremely flexible to fit exactly whatever data the algorithm sees.
- An example of an algorithm that tends to have high variance is a decision tree (especially decision trees with no early stopping parameters).
- A decision tree might attempt to split every point into its own branch if possible.

Linear models have high bias, low variance
Decision trees (no early stopping) have high variance, low bias

Ensemble methods
- Join models to get a better model
- Learn about 2 ensemble methods: bagging (bootstrap aggregating), boosting
- Bagging: models predict separately then the results are aggregated
    - Regression - continuous target - aggregate by averaging
    - Classification - categorical target - aggregate by majority vote
- Boosting: exploits models better
    - Continuously build models correcting errors made by previous models
- Models = week learner (should at least slightly do better than random guessing)
- Combined model = strong learner
- The default for most ensemble methods is a decision tree in sklearn
- By combining algorithms, we can often build models that perform better by meeting in the middle in terms of bias and variance.
- There are some other tactics that are used to combine algorithms in ways that help them perform better as well. These ideas are based on minimizing bias and variance based on mathematical theories, like the central limit theorem.

- Another method that is used to improve ensemble methods is to introduce randomness into high variance algorithms before they are ensembled together.
- The introduction of randomness as in random forests combats the tendency of these algorithms to overfit. There are two main ways that randomness is introduced
    - Bootstrap the data - that is, sampling the data with replacement and fitting your algorithm to the sampled data (efficient for large datasets). Some data points might not be used for training due to random sampling
    - Subset the features - in each split of a decision tree or with each algorithm used in an ensemble, only a subset of the total possible features are used.
- When doing majority votes with even number of models we can pick any way to break ties (ties are less likely for large data and large no. of models)
- Adaboost algorithm
    1. Assign equal weights e.g. 1 to all of data points
    2. Fit model by minimizing total weights of incorrectly predicted points
    3. Find total weights for
        - Correctly predicted points
        - Incorrectly predicted points
    4. New weight of incorrectly predicted points
       = sum(weights of correctly predicted points) / # incorrectly predicted points
       To make total weights of both correctly and incorrectly predicted groups equal
    5. Repeat from 2. stop when certain no. of models are built
    6. Combine model using weights
        - Correct model will get large positive weight
        - Random model (correct half of the time) will get zero weight
        - Always incorrect model will get large negative weight (to do the opposite)
        - Weak learner weight
          = ln(sum(weight correctly predicted) / sum(weight incorrectly predicted))

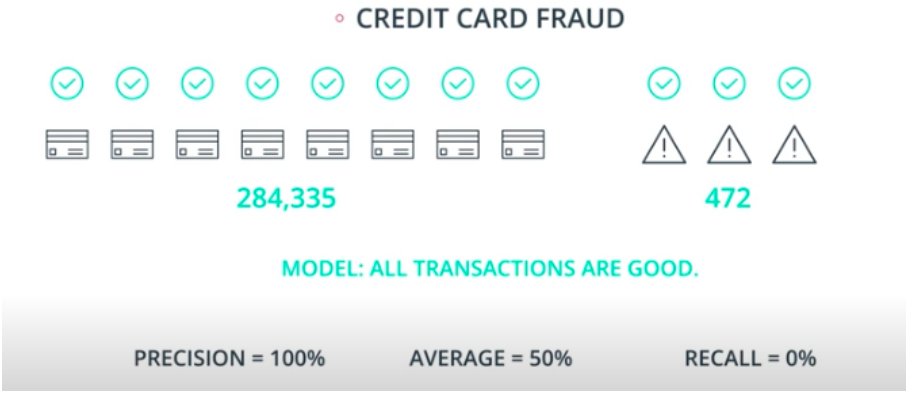


- Adaboost hyper-parameters
    - base_estimator: The model utilized for the weak learners (Warning: Don't forget to import the model that you decide to use for the weak learner)
    - n_estimators: The maximum number of weak learners used (large n_estimators may take longer to train sometimes)
- Bagging is good for combining high variance models due to randomness
- Boosting is good for combining weak models
- Ensemble methods can help optimize for both variance and bias

Evaluation metrics

- Help tell how well the model is doing and help guide how to tweak the model. To know which algorithm works best on data, we need to measure algorithm performance
- Greater generalization ability (or performance on unseen data) make a model better
- Good model should treat outliers as noise (ignore them) and fit data in general way
- Never use testing data for training
- Classification evaluation metrics
    - Confusion metric
        - TP = predicted positive, actual positive
        - TN = predicted negative, actual negative
        - FP = predicted positive, actual negative (Type 1 error)
        - FN = predicted negative, actual positive (Type 2 error)
    - Accuracy
        - Out of all examples how many examples were correctly classified
        - Accuracy = (TP + TN) / (TP + TN + FP + FN)
        - Often asked as percentage (a number between 0 and 100)
        - Accuracy is not always good especially for imbalanced class (skewed class) data e.g. classification on data with class ratio 1:99 can give 99 % accuracy classifying all examples as majority class
    - Precision
        - Out of predicted positives how many are actually positive (predicted correctly)
        - Good for evaluating model that need to avoid FP e.g. spam classifier
        - Precision = TP / (TP + FP)
    - Recall

    - Out of actually positive how many are correctly predicted as positive
    - Good for evaluating model that need to avoid FN e.g. medical model
    - Recall = TP / (TP + FN)
- F1 score
    - Combines precision and recall into one score by harmonic mean
    - F1 score should be low if either precision or recall is low
    - F1 score = 2 * precision * recall / (precision + recall)
    - F1 score is affected by lower number which make it close to the lower number
    - F1 score give equal importance to both precision and recall
    - Why not use arithmetic mean is because F1 score can be misleading for skewed class data
    - The model below has 100 % precision because TP = # predicted positive = 0 which make F1 from arithmetic mean high though recall is 0 %



    - Harmonic mean is always less than arithmetic mean
    - Harmonic mean is affected by lower number which make it close to the lower number

- F-beta score

$$F_\beta \text{ SCORE} = (1+\beta^2)\,\frac{\text{Precision} * \text{Recall}}{\beta^2 * \text{Precision} + \text{Recall}}$$

    - Combine precision and recall into one score but have flexibility for giving more importance to one of precision or recall
    - Good if we want a single number skewed towards one of precision or recall
    - beta is a hyper-parameter
        - Beta can have value from 0 to infinity
        - Small beta (zero or above but less than one) - give more importance to precision
        - Large beta (more than 1) - give more importance to recall
        - Beta = 1 - give F1 score i.e. equal importance to both
        - Beta = 0 - give precision
        - Beta = infinity - give recall

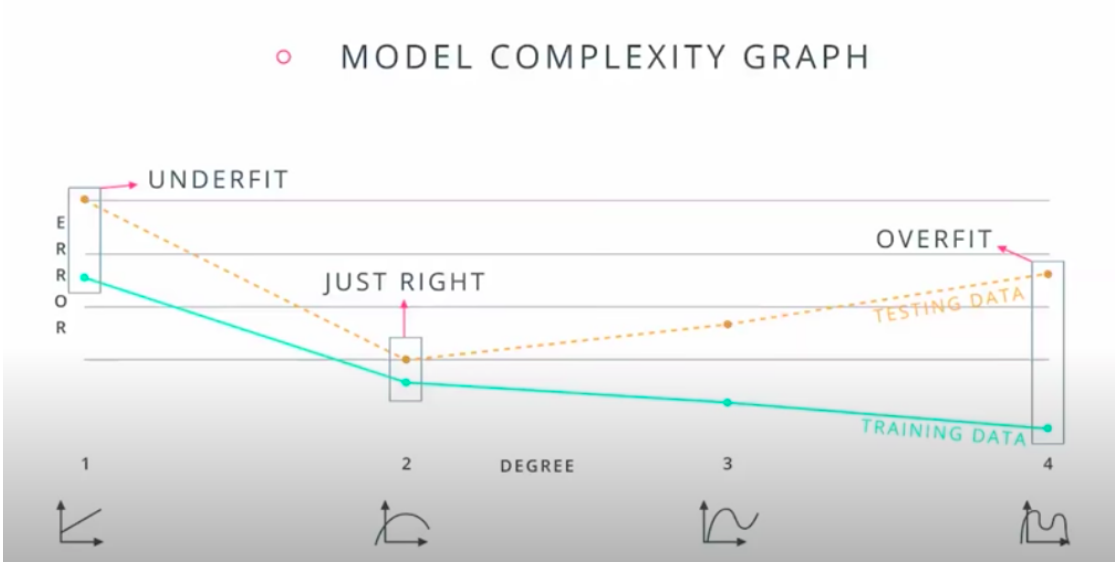
- Area under ROC curve (AUC)
    - TP rate = TP / actual Positive = TP / (TP + FN) = recall

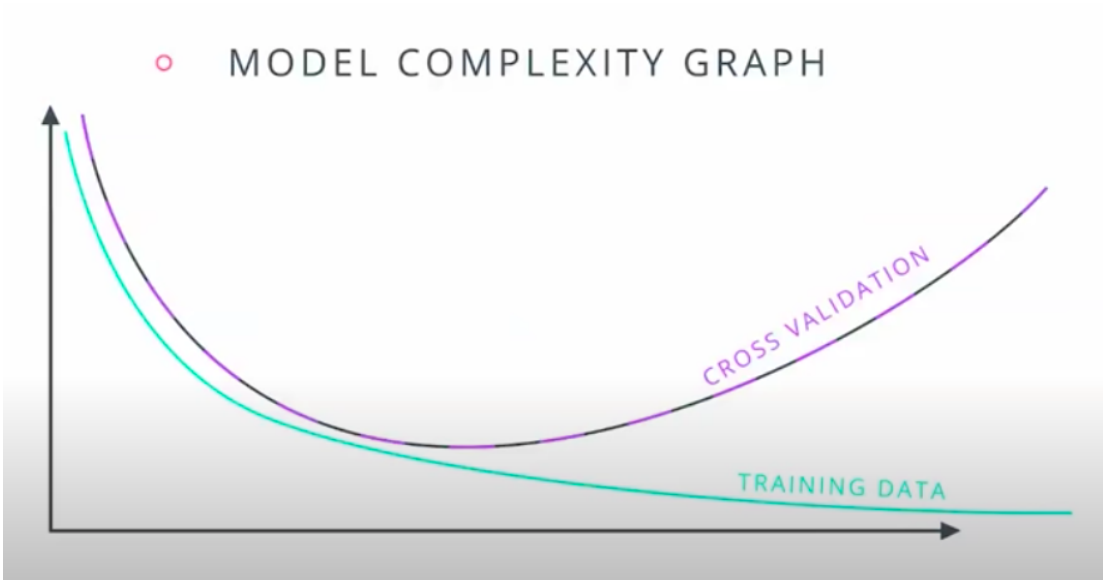
        - FP rate = FP / actual negative = FP / (TN + FP)
        - Plot TP rate on y-axis and FP rate on x-axis
        - Classifying all examples as negative class (very high cut off threshold) will result in TP rate = FP rate = 0
        - Classifying all examples as positive class (very low cut off threshold) will result in TP rate = FP rate = 1
        - Area under ROC curve range from 0 to 1 (close to 1 is better)
        - Square shape ROC curve gives area of 1 (the best). Square ROC curve definitely have TP rate = 1 and FP rate = 0 at some cut off threshold
        - A random model will have AUC of 0.5
        - Model performance is highly affected by quality of data
        - Too small cut off threshold will lead to higher recall, lower precision
        - Too large cut off threshold will lead to higher precision, lower recall
    - Recall on negative class = specificity = TN / (TN + FP) = 1 - FP rate
- Regression evaluation metrics
    - MAE
        - MAE = mean(abs(y - ypred))
        - Range from 0 to infinity, the lower the better
        - Not differentiable (not suitable for gradient descent)
        - useful metric to optimize on when the target follows a skewed distribution because outliers will not influence models as much as if you use the mean squared error
        - The optimal value for MAE is the median value
    - MSE
        - Range from 0 to infinity, the lower the better
        - MSE = mean((y - ypred) ** 2)
        - Differentiable and suitable for gradient descent
    - R2 score
        - Compare model built to a simple model (model that always give mean of target)
        - R2 value is frequently interpreted as the amount of variability in target captured by model
        - R2 score = 1 - (MSE(model) / MSE(simple model))
        - R2 can range from -infinity to 1
            - R2 of 0 means that model is no better than simple model (MSE of model and simple model are equal)
            - R2 of 1 means MSE(model) is a lot smaller than MSE(simple model)
            - R2 less than zero means model is worse than simple model (model has more MSE than simple model)
    - Optimizing on the mean squared error will always lead to the same best model as if you were to optimize on the r2 value
    - model with the highest R2 ia the model with lowest MSE
    - The optimal value of MSE and R2 is actually the mean
    - Best model from minimizing MAE is not necessarily same as the best model from optimizing MSE and R2

Training and tuning

- Underfitting
    - oversimplifying problem (error due to bias)

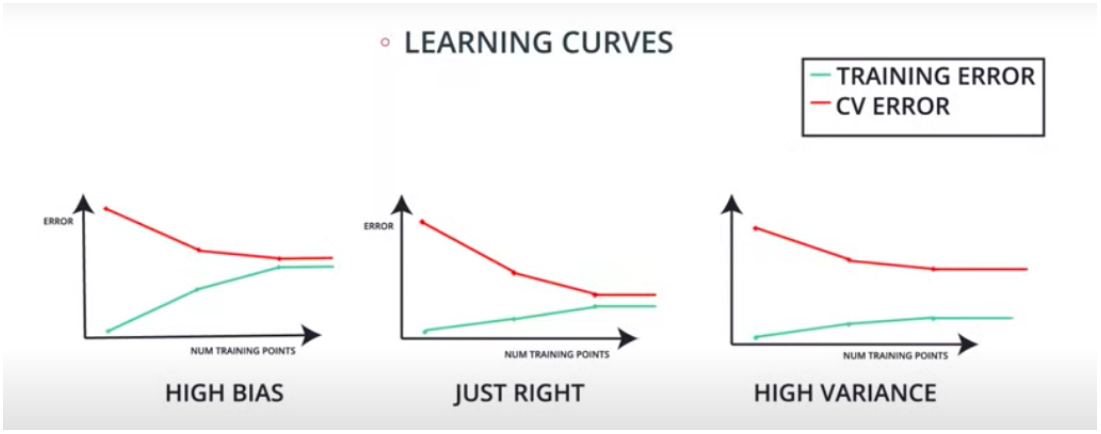
    - Model doesn’t do well on training data and testing data
- Overfitting
    - overcomplicating problem (error due to variance)
    - Model doing very well on training data but not doing well on testing data





- Model complexity graph
    - x-axis = model complexity hyper-parameter
    - y-axis = error (training, validation)
    - Never use testing data to select hyper-parameter

- K-fold cross validation = recycling data or using all parts of data for training
    - Determine K
    - Split data into K equal parts
    - Train K models with different K - 1 parts and validate with 1 part (different for each model)
    - Average the score from K models built
    - Should shuffle data before splitting data into K parts to randomize order of examples and remove any hint of bias



- It doesn't mean that the error is 1 minus the score. It only means that as the model gets better, the error decreases, and the score increases.
- It's always good to do a reality check when we can, and see that our models actually do have the behavior that the metrics tell us.
- Parameters are parameters which value change during training e.g. coefficients of curve, threshold of decision trees
- Hyper-parameters are parameter that don’t change from training
- Grid search in sklearn tries every possible combination of hyper-parameter values specified
- Sklearn grid / random search object need parameter values and scorer