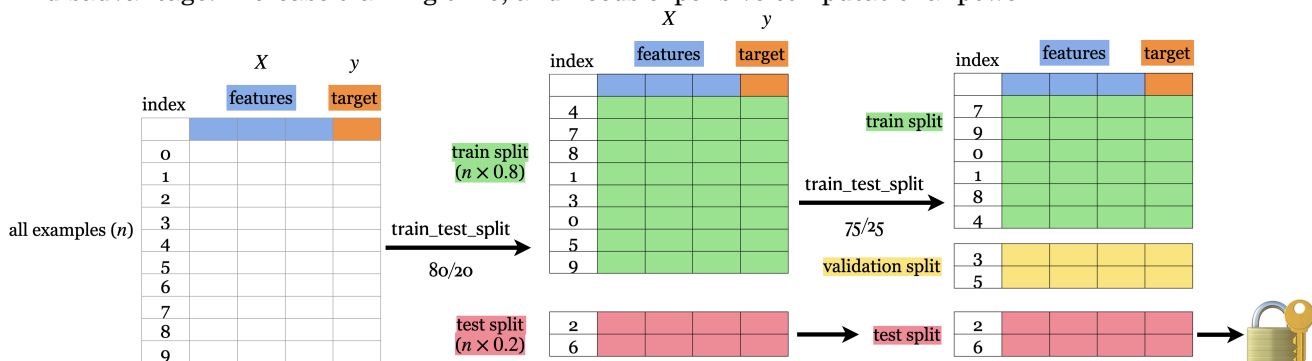


Chapter 1 & 2: Introduction and Decision Trees

- types of machine learning
 - supervised machine learning: given a set of observations (X) and their corresponding targets (y), training a model that relates X to y to predict new X s
 - unsupervised machine learning: train a model to find patterns in a dataset (typically an unlabeled one), an example would be clustering
 - Recommendation systems: predict the "rating" or "preference" a user would give to an item
- classification vs regression:
 - classification: predicting among two or more discrete classes
 - regression: predicting a continuous value
- we use baselines as a sanity check
- Decision Trees:
 - are models that make prediction by sequentially look at features and checking if they're above/below a threshold
 - each node represents a question or answer. The leaf nodes represent answers
 - fit looks to minimize impurity at each question (using the gini index)
 - it considers all features at a particular node and decide which one is the most "important" to split on
 - can be used for continuous values (so regression) - use MSE instead of gini index
 - hyperparameter: `max_depth` → correlates with decision boundaries

Chapter 3: ML Fundamentals

- splitting data is done so we can test how well our model is doing
 - there's test and train set for training and testing (want to use test set once to evaluate performance of best performing model on validation set)
 - there's also validation data for hyperparameter tuning
- cross validation runs validation many time and give each fold a chance to be the validation set, avoid the scenario where you end up with a split that doesn't align/well represent your data
 - advantage: Can assess model and reduce overfitting, can tune hyperparameter to get maximal performance
 - disadvantage: Increase training time, and needs expensive computational power



- overfitting: if model is very complex, then you'll learn unreliable patterns to get every training example correct
 - increasing complexity of models = more chance of overfitting
 - bad because the model is now resistant to new data
 - sign: when training error is low but big gap between training error and validation error
- underfitting: the model isn't learning/performing well enough on training data
- we want the model to be generalize-able to unseen data
- there are no perfect way to pick a hyper-parameter but most common way is to pick model with minimum cross-validation error (or highest cross-validation accuracy score)
- golden rule:** test data cannot influence training phase in any way

Chapter 4: k -NNs & SVM-RBF

- k-Nearest Neighbours:
 - given some data points and their classes, we plot the data points and categorize them by colours or smt. Given a new data point, we grab the k nearest neighbour (via Euclidean distance) and see the majority class
 - # of features = d-dimensions
 - hyperparameter: `n_neighbors` → how many closest points do we consider
 - increasing `n_neighbors` increases model complexity (overfitting)
 - most of the work is done during the predict stage (rather than the fit stage)
 - kNN is very bad with big number of dimensions, as well as when there are irrelevant attributes
- parametric vs non-parametric:
 - an algorithm is non-parametric if it needs to store $O(n)$ worth of stuff to make predictions
 - parametric: **linear SVM RBF**

- non-parametric: k -NN is non-parametric (stores every data point to do distance calculations during predict)
 - SVM RBF is also non-parametric
- SVM-RBF: another popular similarity-based algorithm
 - they're like weighted kNN
 - decision boundary is defined by a set of positive and negative examples and their weights
 - unlike kNN, SVM RBF only remember the key examples (support vectors) - so runs faster
 - usually perform better too
 - hyperparameter: γ and C
 - larger γ = more complex
 - larger C = more complex as well
- both kNN and SVM RBF has a regressor counterpart

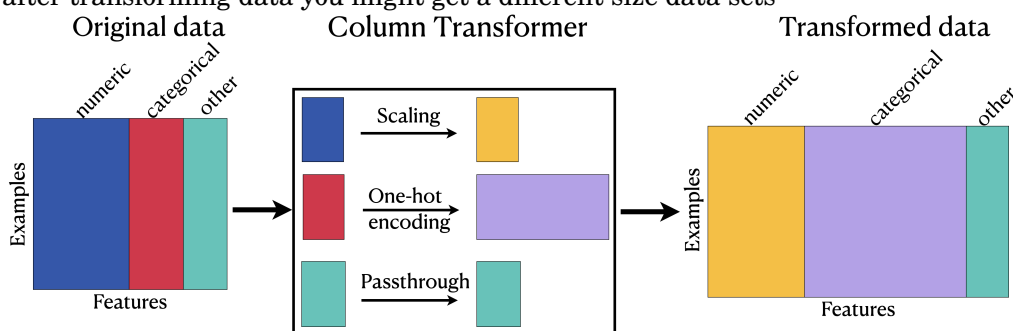
Chapter 5: Preprocessing Pipelines

- when working with numeric data on different scales, and as a result features with larger values (scale) tends to dominate and smaller scale features get ignored
 - we use a transformer: StandardScaler
- fit and transform paradigm
 - split the data before you do pre-processing
 - fit_transform the transformer on training set
 - call transform (only) on the test set
- imputation: tackle missing values
 - SimpleImputer is the transformer we'll use
 - common strategy is to replace missing values with most frequent value in the categorical columns, and fill in the median value in the numeric columns
- one-hot encoding (OHE): tackle categorical values
 - we can transform categorical features into numeric ones so we can use them (assign an integer to each unique categorical label)
 - OneHotEncoder creates binary columns to represent our categories (c categories in column results in c new binary columns)
 - handle_unknown = "ignore" parameter gets it to ignore unknown value (happens when you have a low number of a category and it ends up in validation set and none in the training set - see Chapter 6 → dealing with unknown categories)
 - drop = 'if binary' is used for binary categories, it'll create 1 column of 0s and 1s instead of make 2 columns
- ordinal encoding: when there is an order to the categorical features
 - ex. Excellent → Good → Average → Poor
 - see code in Chapter 6 → incorporating ordinal features
 - when you have more than one ordinal columns, you can pass a list of lists to OrdinalEncoder where the inner list corresponds to the ordered categories for corresponding ordinal column (ex. [[Great, Okay, Bad], [Smart, Average, Stupid]])
 - whether you go best to worst or worst to best doesn't matter
- to do pre-processing and CV and not break the golden rule, you're gonna need pipelines
 - you can call fit on the pipeline and it'll run through all the steps for you
 - can also call predict

Chapter 6: Column Transformer and Text Features

Column Transformers

- use ColumnTransformer when you want to apply different transformers to different rows
 - i.e scaling for numeric features and OHE for categorical features
- call fit_transform on the training set
- you can pass column transformers into pipelines
- after transforming data you might get a different size data sets



- ex. pipe = make_pipeline(ct, SVC())

Encoding Text Features

- BOW representations
 - ignores the syntax and word order
 - 2 components
 - the vocabs (all unique words in the documents)
 - value indicating either the presence/absence or the count of each word in the document
 - with CountVectorizer you need to define separate CountVectorizer transformers for each text column, if you have more than one text columns.
 - see lecture 6 for hyper-parameters

Chapter 7: Linear Models

- linear models is a fundamental and widely used class of models. They are called linear because they make a prediction using a linear function of the input features
- Linear regression
 - each feature is assigned a coefficient and the model learns an intercept, given new data points, you can plug in the multiply by the weights and add them up for the prediction
 - when we call fit, a coefficient/weight is learned for each features and they're learned from the training data
 - positive coefficient means proportional: bigger feature = bigger target
 - magnitude: bigger magnitude means bigger impact on prediction
 - when interpreting coefficients, scaling is crucial
 - in linear models for regression, the model is a line for a single feature, a plane for two features, and a hyperplane for higher dimensions → so we'll have a k -dimensional plane for k features
- Ridge
 - is another linear regression model with a complexity hyper-parameter alpha
 - bigger alpha means underfitting (alpha being 0 is the same as using Linear Regression)
- Logistic Regression
 - **is a linear model for classification**
 - it outputs a raw score, and based on a threshold (i.e raw score > 0.50), it will decide on whether the class is positive or negative
 - raw score below the threshold would be predicted as a negative class
 - also has coefficients (interpretation is similar to linear regression)
 - model usually randomly considers one class to be "positive" or "negative"
 - classes_ attribute tells us which class is considered negative and positive
 - is off the form [negative, positive]
 - hyperparameter: C - bigger C means overfitting
- predict_proba
 - is a soft prediction, i.e how confident a model is with a given prediction
 - you can call predict_proba and it'll give you [probability it's the negative class, probability it's the positive class]
 - these values are achieved by feeding the raw score (weighted sum) into the sigmoid function
 - points closer to the boundary means model is less confident
- see chapter 7 on using LR with words

Chapter 8: Hyperparameter Optimization and Optimization Bias

Hyperparameter Optimization

- manual hyperparameter optimization like we've been doing takes a lot of work, not reproducible, and in complicate cases (i.e multiple hyper parameter) → intuition might be worse
- GridSearchCV:
 - we need
 - instantiated model or pipeline
 - parameter grid: user specified set of values for each hyperparameter that we're gonna look through
 - other optional argument (i.e n_jobs = -1 which means use all cores)
 - the method considers every combination of the value sets and test out each one
 - you can call fit, predict, or score on it and call best_score_ and best_params_ attributes
 - the range of which we pick the hyperparameter to be in play an important results
 - **computation**: if you have 5 hyperparameters, and 10 different values for each hyperparameters, you will have to evaluate 10^5 models (not counting number of cv folds)
- Randomized Hyperparameter Search
 - you specified a list of values for each hyperparameter, similar to above
 - difference is this one picks random combinations to try out
 - hyperparameter: n_iter tells it how many models to try out
- note that the exponential range for the hyperparameter C is quite common (usually 10^n where $n = -2, -1, 0, 1, 2, \dots$)
- advantages of Randomized Hyperparameter Search

Optimization bias/Overfitting of the validation set

- when our dataset is small and if your validation set is hit too many times, we suffer from optimization bias or overfitting of the validation set
- optimization bias of parameter learning
 - basically overfitting with training error
- optimization bias:
 - when training data is small and so is the validation splits, we could hit the same validation set many times and that validation might have given us good results
 - so like the the small validation set could have just been really good but this model keeps using it over and over so we keep getting this great score over and over
 - even though validation splits will not influence training directly - it will influence hyperparameter optimization
- so we can trust validation scores when the training data is of good size, and we can trust the test score score when the test set is of good size

Chapter 9: Classification Evaluation Metrics

- when we have an unbalanced dataset, accuracy will not be a good metrics
- when we're trying to spot a certain label (i.e fraud) → this is called the positive class
- confusion matrix
 - false positives (type 1 errors): model incorrectly spots examples as fraud
 - false negatives (type II errors): it didn't classify fraud examples as fraud (classified them non-fraud instead)
 - diagonal entries are the perfect predictions while the off diagonals tell us what is being mis-interpreted
 - note that the order of the cells could be different sometimes (though the diagonal thing is always true)

true not Fraud	59700	8	true not Fraud	TN	FP
true Fraud	38	64	true Fraud	FN	TP
	predicted not Fraud	predicted Fraud		predicted not Fraud	predicted Fraud

- other scoring metrics for when there's a class imbalance and we can't use accuracy
 - recall: among all positive examples, how many did you identify

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

- precision: among the positive examples you identified, how many were actually positive?

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

- f1-score: combines precision and recall to give one score - good for hyperparameter optimization

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

- you can use `classification_report` to get this, the rows are "what if this class was the positive class"

- **when you're trying to avoid false-negatives: use recall**

- ex. false negatives are very bad in cancer diagnosis - we'd rather falsely diagnose someone and give them tests than to let them go when they have cancer

- **when cost of a false-positive is high: use precision**

- ex. you're a restaurant looking to buy wine only if it's been predicted as good by a classifier - since cost is limited, we don't want to waste money on shitty wine, we rather miss out on some good wines but all the ones we buy to be good

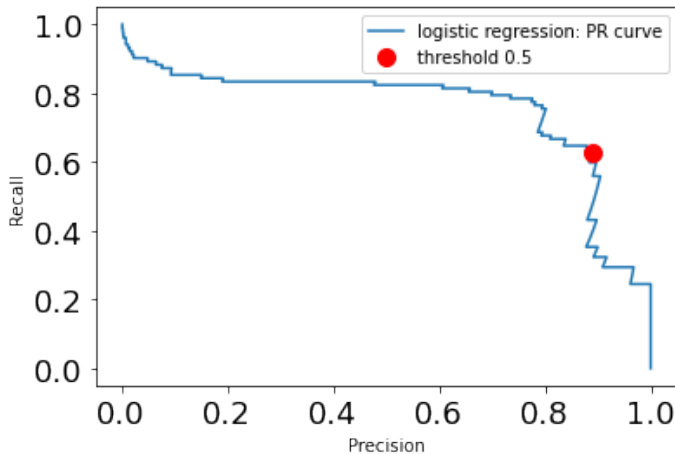
- accuracy and these new metrics are not very correlated (i.e model A having higher accuracy than model B doesn't mean it has higher precision) - there is a relationship between precision and recall however

- operating point: confusion matrix uses hard predictions, here we're leveraging the confidence (`predict_proba`) of the model to understand the model performance

- if you want to achieve a certain percentage for the "fraud class", you can change the threshold of `predict_proba` (so you can make it smaller to identify more positive examples)
- setting a requirement on a classifier (e.g recall of ≥ 0.75) is called setting the **operating point**

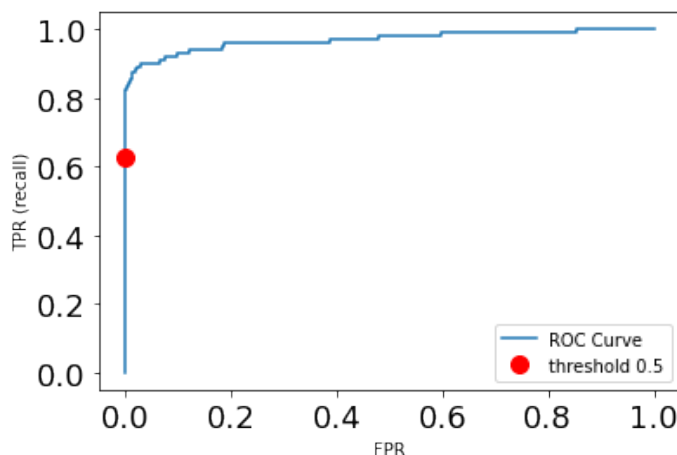
- precision/recall tradeoff
 - there's a trade off between precision and recall
 - if you identify more things as "fraud", recall is going to increase but there are likely to be more false positives
 - decreasing the threshold means lower bar for predicting fraud
 - you are willing to risk more false positives in exchange of more true positives.
 - recall would either stay the same or go up and precision is likely to go down
 - point: decreasing threshold likely to increase recall and decrease precision
 - 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 as the denominator for precision is TP+FP.
 - point: increasing the threshold usually means recall goes down and precision goes up

- PR (precision-recall) Curve



- the red dot correspond to the threshold we specified for the `predict_proba`
 - so, we can achieve a recall of 0.8 and precision of 0.4 when the threshold is 0.5
 - here we have a high precision but lower recall.
 - usually goal is to keep recall high as precision goes up
- AP score: is a number to summarize the PR plot
 - AP (average precision) score: area under the PR curve
 - has value from 0 (worst) to 1 (best)
- AP vs F1 score
 - F1 score is for a given threshold and measures the quality of predict. (default uses threshold `predict_proba > 0.5`)
 - AP score is a summary across thresholds and measures the quality of `predict_proba`.
- ROC curve
 - another commonly used tool to analyze the behavior of classifiers at different thresholds.
 - 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).

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



- ideal curve is to the top left (classifier with high recall while keep low FPR)
- the red dot correspond the threshold of 0.5
- AUC (area under the curve): provides a meaningful number for model performance (based on ROC curve)
 - AUC of 0.5 means random chance
- for classification problems w/ imbalanced classes, AP score or AUC is often much more meaningful than accuracy
- handling imbalance: in this class we'll look into handling imbalances using weights (`class_weights` in sklearn)
 - `class_weight=balance` reduces false negatives but increases false positives higher recall but lower precision
 - `class_weight = {x: y}` reduces false negatives but increases false positives also

- summary:

Confusion Matrix Components

Below are different components of a confusion matrix for a binary classification task with classes **Positive** and **Negative**.

		Predicted		
		Positive	Negative	Total
Actual	Positive	True positive (TP)	False negative (FN) (Type 2 error)	# positives
	Negative	False positive (FP) (Type 1 error)	True negative (TN)	# negatives
Total		TP + FP	FN + TN	# examples

Confusion Matrix Example

		Predicted		
		Positive	Negative	Total
Actual	Positive	80	40	120
	Negative	20	60	80
Total		100	100	200

Accuracy and Error

$$accuracy = \frac{TP+TN}{TP+FP+TN+FN} = \frac{TP+TN}{\#examples}$$

$$error = \frac{FP+FN}{TP+FP+TN+FN} = \frac{FP+FN}{\#examples}$$

Examples

$$accuracy = \frac{80+60}{200} = \frac{140}{200} = 0.70$$

$$error = \frac{20+40}{200} = \frac{60}{200} = 0.30$$

Precision

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

Example

$$precision = \frac{80}{100} = 0.80$$

Recall/TP rate/sensitivity

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

Example

$$recall = \frac{80}{120} = 0.666$$

F₁ score

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

Example

$$F_1 = 2 \times \frac{0.8 \times 0.666}{0.8 + 0.666} = 0.727$$

True Negative Rate (specificity)

$$tnr = \frac{TN}{\#negatives}$$

Example

$$specificity = \frac{60}{80} = 0.75$$

False Positive Rate

$$fpr = \frac{FP}{FP+TN} = \frac{FP}{\#negatives}$$

Example

$$fpr = \frac{20}{80} = 0.25$$

False Negative Rate

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

Example

$$fnr = \frac{40}{120} = 0.333$$

Chapter 10: Regression Evaluation Metrics

- beginning of chapter 10 has quite a nice workflow if you need to see code
- regression score functions: since we're not doing classification anymore, we can't just check for equality, some common metrics are
 - mean squared error (MSE)
 - R^2
 - root mean squared error (RMSE)
 - MAPE
- mean square error (MSE)
 - perfect predictions have $MSE = 0$
 - the score depends on the scale of our targets (and units) - so scores can get a big and unwieldy
 - also, the unit of the score is different (target is in dollars, MSE is in dollar²)
- root mean square errors (RMSE)
 - a more relatable metric
 - is the square root of the MSE
- R^2
 - is used by default by sklearn when you call `score()`
 - max value is 1 (for perfect predictions)
 - values can be negative (worse than dummy regressor - very bad)
 - sklearn uses the negative version of this (negative root mean squared error), and it does this because it wants to think of it in the context of maximizing (so higher neg RMSE is better)
- MAPE
 - takes into account the relative error (i.e 30k error for a 600k house is reasonable, but for a 60k house, it's awful)
 - it looks at percent error and takes the average over all examples
 - ex. if we get MAPE score of 10.09, we can interpret as "on average, we have around 10% error"
- transforming the target
 - linear regression models by default try to minimize RMSE, but sometimes we'd like it to minimize MAPE
 - common practice to get `.fit()` to care about MAPE is to log transform the targets - so $y \rightarrow \log(y)$
 - we log transform when there are extreme values and when we want interpretability in terms of units

- you can also use these scoring functions with `cross_validate`, by passing them into the `scoring = "(insert metric)"`

Chapter 11: Ensembles

- ensembles: models that combine multiple ML models to create more powerful models
- tree-based ensemble models
 - decision trees are interpretable and can capture non-linear relationships, but likely to overfit → idea is to combine multiple trees to make a stronger model
- `RandomForestClassifier`
 - hyperparameter:
 - `n_estimators`: tell it how many trees we want to build (higher = more complexity)
 - `max_depth`: max depth of each decision tree (higher = more complexity)
 - `max_features`: number of features to look at at each split (higher = more complexity)
 - fit a diverse set of many decision trees by injecting randomness into the classifier construction
 - predicting by voting (classification) or averaging (regression) the predictions given by individual trees
 - injecting randomness:
 - data: Build each tree on a bootstrap sample (i.e., a sample drawn with replacement from the training set)
 - ex. Suppose this is your original dataset: [1,2,3,4]
 - a sample drawn with replacement: [1,1,3,4]
 - a sample drawn with replacement: [3,2,2,2]
 - a sample drawn with replacement: [1,2,4,4]
 - features: at each node, select a random subset of features (controlled by `max_features` and we select new subset at every node) and look for the best split involving one of these features
 - ensembles seem to beat the fundamental tradeoff - we can increase the training score while at the same time not decreasing validation score so much
 - pros:
 - usually one of the best performing off-the-shelf classifiers without heavy tuning of hyperparameters
 - don't require scaling of data - all tree based model not that sensitive to scale of data
 - less likely to overfit
 - In general, able to capture a much broader picture of the data compared to a single decision tree.
 - cons:
 - require more memory
 - hard to interpret
 - tend not to perform well on high dimensional sparse data such as text data
- Gradient Boosted Trees
 - ex. `XGBoost`, `LightGBM`, `CatBoost`
 - there are no randomization
 - idea so to combine many simple models (weak learners) to create strong learners
 - they combine multiple shallow (depth 1-5) decision trees
 - build trees in a serial manner, where each tree tries to correct mistakes of previous ones
 - hyperparameter:
 - `n_estimators`: number of trees to build
 - `learning_rate`: controls how strongly each tree tries to correct mistake of previous ones (higher learning rate = more complex models)
- which model to pick: pick the one that has the best CV score, but also take into consideration CPU cost
- Averaging: use a bunch of models, predict for each example, and use the majority or average
 - uses `VotingClassifier`
 - as long as different models make different mistakes, this would work

Example	log reg	rand forest	cat boost	Averaged model
1	✓	✓	✗	✓✓✗=>✓
2	✓	✗	✓	✓✗✓=>✓
3	✗	✓	✓	✗✓✓=>✓

- cons:
 - takes a long time
 - reduction in interpretability
 - reduction in code maintainability
- Stacking:
 - instead of averaging the outputs of each estimator, we use their outputs as inputs to another model
 - the final estimator by default, for classification is logistic regression
 - it takes in predictions (`predict_proba` scores) of other classifiers for each example
 - so number of coefficients (and number of features) = number of base estimators

- the coefficients we get back from LR can be interpreted as which base model LR thought was the most important in predicting a class
- usually takes longer than voting, but generally have higher accuracy (you can also see the coefficients)

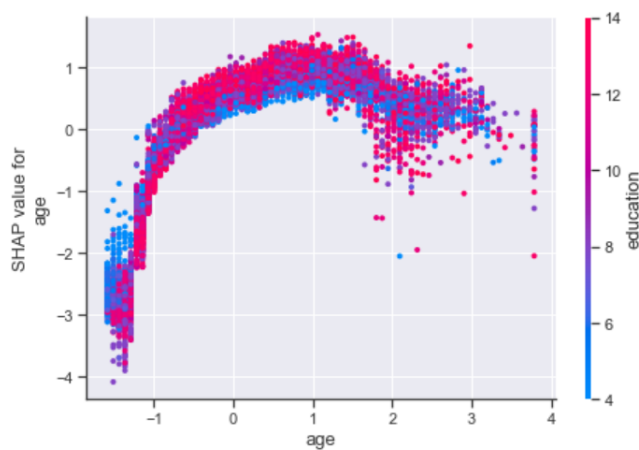
Chapter 12: Feature Importance

- during EDA, we can look at correlation between various features with other features and the target in our encoded data
 - see notebook to see how to call heat maps
 - big positive values in cells mean X and Y is highly correlated
 - this approach is too simplistic though
 - only looks at features in isolation and linear associations
 - ex. SalePrice is deemed uncorrelated with BsmtFullBath but it may be the case that SalePrice is high when BsmtFullBath is 2 or 3, but low when it's 0, 1 **or** 4 (and hence it'll seem uncorrelated)
 - it can give us a good hint about high correlation - but don't take the uncorrelated measurements too seriously
- interpretation of correlation
 - if Y goes up when X goes up, we say X and Y are positively correlated
 - if Y goes down when X goes up, we say they're negatively correlated
 - if Y is unchanged when X , we say they're uncorrelated
- feature importance in linear models: for linear models (i.e logistic regression and linear regression, we can look at coefficients for each features, let's look at how to interpret them
 - **ordinal features**: how much the predicted target (i.e SalePrice) increase/decrease by the feature (i.e ExteriorQuality going up one category (i.e good → excellent)
 - **categorical features**: each category gets their own coefficients, we can talk about the change in prediction from switching one to another by picking a "reference" category, and subtract everything by that

	Coefficient	<code>lr_coefs_landslope - lr_coefs_landslope.loc["LandSlope_Gtl"]</code>	
LandSlope_Gtl	457.197456		
LandSlope_Mod	7420.208381		
LandSlope_Sev	-7877.405837		

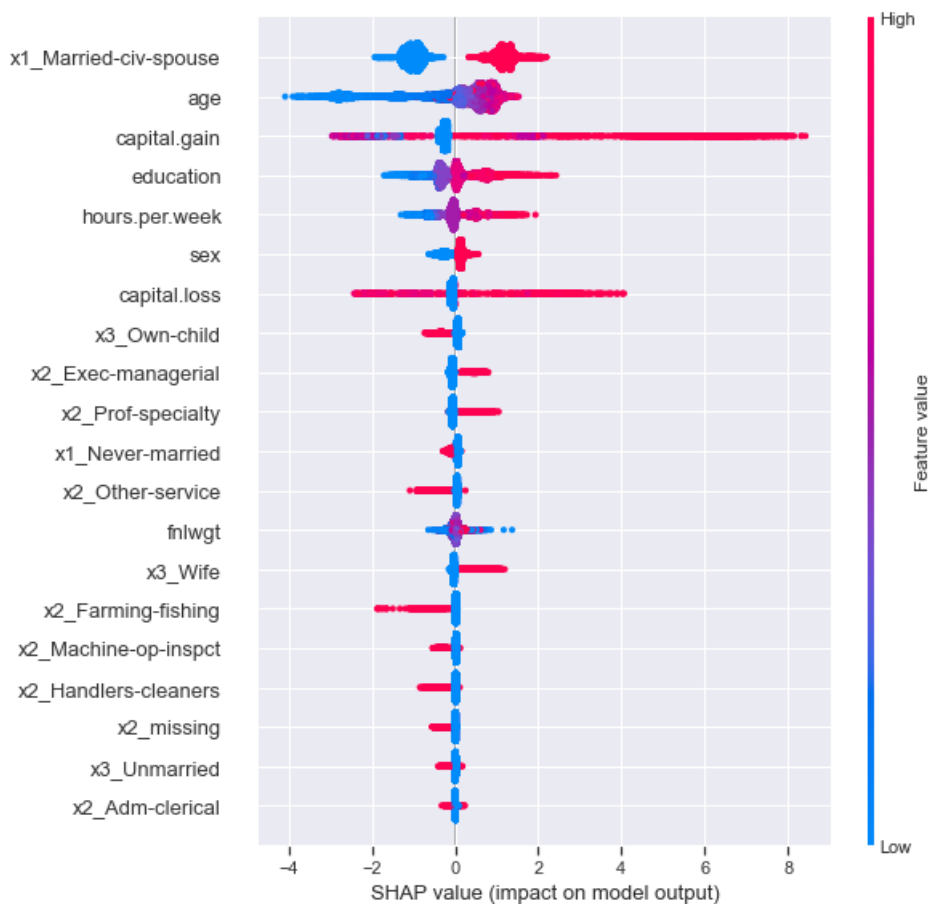
	Coefficient
LandSlope_Gtl	0.000000
LandSlope_Mod	6963.010925
LandSlope_Sev	-8334.603292

- ex. we have the following coefficients for the categories inside LandSlope. This means that you change category from LandSlope_Gtl to LandSlope_Mod the prediction price goes but by \$6963
- numeric features: is tricky because numeric features are scaled (calling `coefs` alone will get you weird numbers that might not be correct to scale)
 - we needed to access the scaler and unscale it
 - once you do that, the coefficient that if we increase the **scaled** feature by one unit the price would go up z amount
- interpretability: ability to interpret ML models is important - we can leveraged by domain experts to diagnose systematic errors and underlying biases
 - simple models are more interpretable but not as accurate
- interpretability and feature importance beyond linear models
 - sklearn has package `features_importances_` that gives feature importance for sklearn's tree based algorithms
 - unlike linear model coefficients, `features_importances_` doesn't have a sign
 - they tell us about importance (how important it is to the model's prediction) but not "up" or "down"
 - because increasing a feature may cause prediction to go up, then down (not possible in linear models)
 - eli5: used to get feature importance for non sklearn models - you get feature importance coefficients just like you would for sklearn's model
 - SHAP: sophisticated measure of contribution of each feature
 - there will be SHAP value for each example and each feature - the plots look at the average of these
 - dependence plot: x -axis is a feature (age here), y -axis is SHAP value



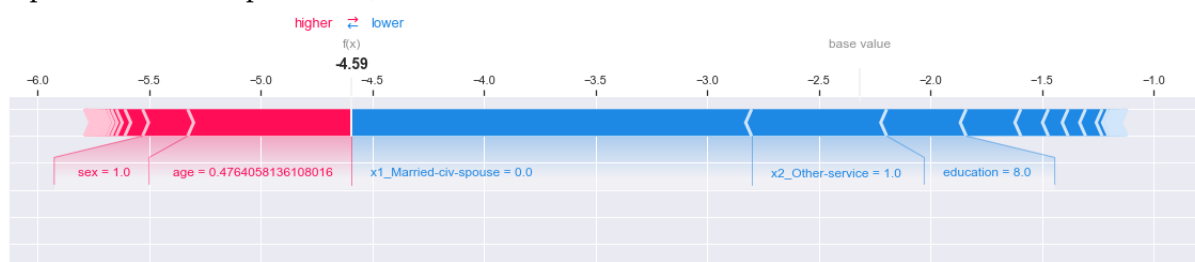
- this is a plot for training dataset
- we can see that smaller age leads to smaller SHAP value (means less likely to predict the positive class)
- we can also notice that non-linear relationship, so middle-aged people are more likely to get predicted positive class rather than very young and very old people

→ summary plots: shows overall impact for all features



- colours here represent SHAP value (red being high SHAP value)
- so we can see that higher capital gain means a bigger SHAP value

→ force plot: how are the features pushing/pulling away from the expected (avg) for one example (used to justify/-explain the model's prediction)



- this was ran on an example that was a negative class (actual target)
- see how certain things is pushing it back and forward from the average - i.e absence of spouse is pushing it to be more negative, and more negative means we're more likely to predict the negative class

Chapter 13: Feature Engineering and Feature Selection

Feature Engineering

- **Feature Engineering**: process of transforming raw data into features that better represent underlying problem to the predictive models, resulting in improved model accuracy on unseen data
- better features = better models → good features should
 - capture important aspects of problem
 - allow learning with few examples
 - generalize to new scenarios
- trade-off between simple vs expressive features
 - simple features: overfitting risk is low, but scores might be low
 - complicated features: scores can be high, but so is overfitting risk
- best features (decided by the feature) is dependent on model you use → so there's no concept of universal "best features"
- domain specific transformations: there are some natural transformations to do for some domains (i.e image data, sound data, etc) - we'll focus on
 - text data
 - audio data
- common features used in text classification
 - bag of words
 - see above, good for many things
 - the encoding throws out a lot of things we know about language (i.e assumes word order isn't important)
 - to improve scores you carry out feature engineering
 - N-grams
 - incorporates more context
 - likes BOW but takes in a contiguous sequence of n -words in test
 - do this by passing `ngram_range` into `CountVectorizer`
 - increasing `ngram_range` means more likely to overfit
 - Part-of-speech (POS) features
 - a kind of syntactic category that tells you some of the grammatical property of a word
 - it attaches a POS tag to each word
 - we often use pre-trained models to extract POS information (i.e Spacy or nltk) → can even interpret emojis
- classify music style from audio files
 - we'll use `librosa` for feature engineering of audio files
 - much better results with domain-specific features

Feature Selection

- reasons for feature selection
 - interpretability: less features are more interpretable
 - computation: faster fit/predict with fewer columns
 - data collection: cheaper to collect with fewer columns
 - fundamental trade-off: can reduce overfitting by removing useless features
- methods for feature selection
 - use domain knowledge to discard features
 - automatic methods:
 - model-based selection
 - recursive feature-elimination
 - forward selection
- **model-based selection**: use a supervised machine learning model to get feature importance, pick ones that we deem are important, then finally pass the remain features into our final estimator
 - model used for feature selection can be different from one used in final estimator
 - use `SelectFromModel` transformer → can pass into a pipeline (usually after transformer steps)
 - it selects from features which have feature importance greater than provided threshold (so pick features with `predict_proba` score greater than a certain threshold)
- **recursive feature elimination (RFE)**:
 - build a series of model, at each iteration, discard the least important feature according to the model
 - it's like model-based selection many times → computationally expensive
 - basic idea: fit model → find least important feature (just 1 each time) → remove → iterate
 - you will need to decide on k - number of features to select
 - you can use CV (namely `RFECV` to use cross-validation to select number of features) → this is insanely slow (bc it's CV within CV)
- **Search and Score**:
 - define a scoring function $f(S)$ that measures quality of set of features S , now search for best set of features S (where S is can be any combination of the features) - stop when adding/removing a feature doesn't improve the score
 - pick the S with the best score
 - let number of features = n , there are 2^n combinations that you need to try out

- Forward or Backwards Selection (aka wrapper methods):

- shrink or grow feature set by removing/adding one feature at a time
- makes the decision based on whether adding/removing feature improves CV scores or not (below is example of forward selection)

iteration	current round scores	candidates	selected features	best score (error)
1.	$score(x_1) = 0.40$ $score(x_2) = 0.39$ $score(x_3) = 0.43$ $\checkmark score(x_4) = 0.30$	$\{x_1, x_2, x_3, x_4\}$	$\{\}$	∞
2.	$score(x_1, x_4) = 0.35$ $\checkmark score(x_2, x_4) = 0.28$ $score(x_3, x_4) = 0.4$	$\{x_1, x_2, x_3\}$	$\{x_4\}$	0.30
3	$score(x_1, x_4, x_2) = 0.29$ $score(x_3, x_4, x_2) = 0.30$	$\{x_1, x_3\}$	$\{x_4, x_2\}$	0.28
No score is less than the best score (error) so STOP.				

- warning about feature selection

- feature's relevance is only defined in context of other features (i.e adding/removing features can make features relevant/irrelevant)
 - simple association-based feature selection approaches do not take into account the interaction between features
- if features can be predicted from other features, you cannot know which one to pick
- relevance for features does not have a casual relationship

Chapter 14: Clustering

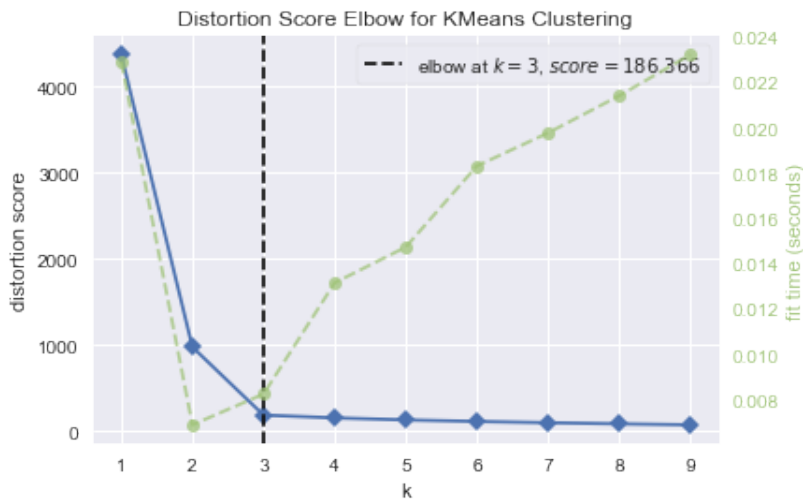
- clustering: task of partitioning dataset into groups called clusters → is unsupervised learning
 - if you have access to labeled training data - you're in "supervised" learning
 - for unsupervised, training data consists of observations (X) with no corresponding targets
 - we can learn without targets but it'll be focused on finding the underlying structure of input
- most intuitive way to get useful information from unlabeled data is to group similar examples together to gain insights
- in clustering, meaningful groups are dependent on the application → makes it hard to measure quality of clustering algorithm
- similarity and distance
 - clustering is based on the notion of similarity or distance between points
 - we'll be using Euclidean distance just like kNN
- K-Means clustering
 - input: X - a set of data points and k (or $n_clusters$) - number of clusters we want
 - output: k clusters (groups) of data points
 - main idea: represent each cluster by its cluster center and assign a cluster membership to each data point
 - hyperparameter: $n_clusters$ - though we can't do CV to determine K , we need to do something else, see below
- K-Means algorithm:
 - input: data points X and the number of clusters k
 - initialization: k initial centers for the clusters
 - iterative process:
 - assign each example to closest center
 - estimate new centers as average of observations in a cluster
 - do this until centers stop changing or maximum iteration reached
 - when centers stop moving - it means that the algorithm has converged → **K-Means always converge (not always to optimal solution)**
 - **because k-Means depends on stochastic initialization of the initial cluster centers - how they're initialized can affect the results big time**
 - note that K-mean is really bad at identifying complex shapes (see lecture 15 K-means recap)
 - every point get assigned to a cluster (even if it's bad)
- Hyperparameter tuning
 - Elbow Method: looks at sum of intra-cluster distances (aka inertia)

$$\sum_{P_i \in C_1} distance(P_i, C_1)^2 + \sum_{P_i \in C_2} distance(P_i, C_2)^2 + \sum_{P_i \in C_3} distance(P_i, C_3)^2$$

C_i are the cluster centers

P_i s are the points in that cluster

Distance is the euclidean distance



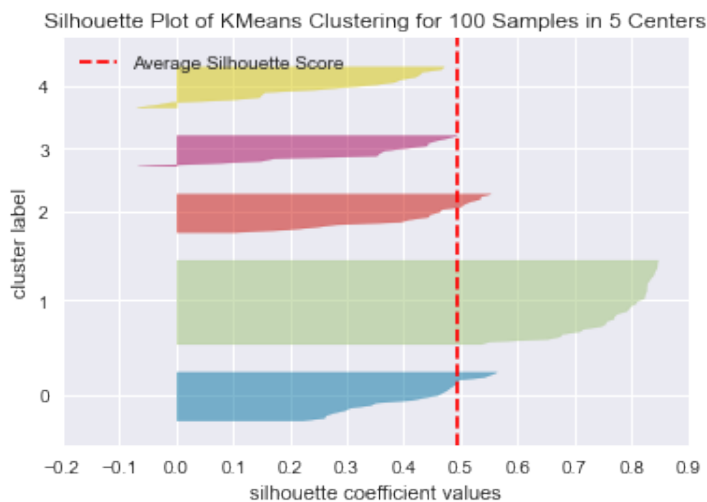
- inertia decreases as K increases
- we want inertia to be small (though picking k to be too big would mean our model is useless)
- do elbow plot and pick the k with big improvement from $k - 1$ but not so much improvement in going to $k + 1$
- can use yellowbrick to make easy elbow plots

○ Silhouette Method: calculated using the **mean intra-cluster distance** (a) and the **mean nearest-cluster distance** (b) for each sample

- mean intra-cluster distance (a):
- mean nearest-cluster distance (b):
- silhouette distance for a sample:

$$\frac{b - a}{\max(a, b)}$$

- best value is 1, worst is -1 (samples has been assigned to wrong cluster)
- value near 0 means overlapping clusters
- the overall Silhouette score is average of Silhouette scores for all samples
- we can plot Silhouette scores for each cluster sample



- size of silhouette shows number of data in the samples (so show imbalance of data points in clusters)
- higher value indicate well separated clusters
- thickness of silhouette indicates cluster size
- shape of each silhouette indicates the "goodness" for points in each cluster
- length (area) of each silhouette shows goodness of each cluster
- slower dropoff (more rectangular) indicates more points are "happy" in their clusters

→ larger values of average Silhouette scores are better

• preprocessing methods like scaling and imputation are unsupervised methods












Chapter 15: DBSCAN and Recommender Systems

DBSCAN

- DBSCAN: Density-Based Spatial Clustering Applications with Noise
 - it's based on the idea that clusters form dense regions in the data - so it works to identify "crowded" regions
 - can address some limitations we saw like:
 - does not require user to specify number of clusters in advance
 - can identify points that are not part of any clusters
 - can capture clusters of complex shape
 - hyperparameters
 - eps: determines what it means for points to be "close"
 - min_samples: determines number of neighbouring points required to consider a point to be a part of a cluster
 - effects of hyperparameters
 - small eps means that if there's just one data point, it'll be considered a cluster (we will get a lot of noise points)
 - large min_samples means that we might consider everything to be a cluster
 - we want something in between (thus we will have to tune the hyperparameters)
 - we cannot call predict (DBSCAN only clusters clusters point you have, not new/test points)
- DBSCAN algorithm: is iterative
 - start with random point
 - check if that point is part of crowded area (if there's enough neighbours)
 - give some colour to that point if so, spread that colour to all its neighbours
 - do the same for all its neighbours
 - when you run out of points, you pick a new random point **that hasn't been visited before**
 - rinse and repeat
- hyperparameter tuning: cannot use elbow method, but can use silhouette method
- weaknesses: doesn't do very well when we have clusters of different densities (K Means can handle these cases quite easily)

Recommender Systems

- recommender system: a system that recommends a particular product/service to users that they're likely to consume
- main approaches:
 - collaborative filtering (most popular)
 - unsupervised learning
 - we have labels y_{ij} (ratings of user i for item j) aka utility matrix
 - we learn the features
 - content-based recommenders
 - supervised learning
 - extract features x_i of users and/or items and building a model to predict rating y_i given x_i
 - apply model to predict for new users/items
 - hybrid: mixed of both above
- Collaborative Filtering
 - given a utility matrix of N users and M items (which is usually sparse), we want to complete it (predict the missing values in the matrix)

					
	Item 1	Item 2	Item 3	Item 4	Item 5
 User 1	?	?	2	?	3
 User 2	3	?	?	?	?
 User 3	?	5	4	?	5
 User 4	?	?	?	?	?
 User 5	?	?	?	5	?
 User 6	?	5	4	3	?

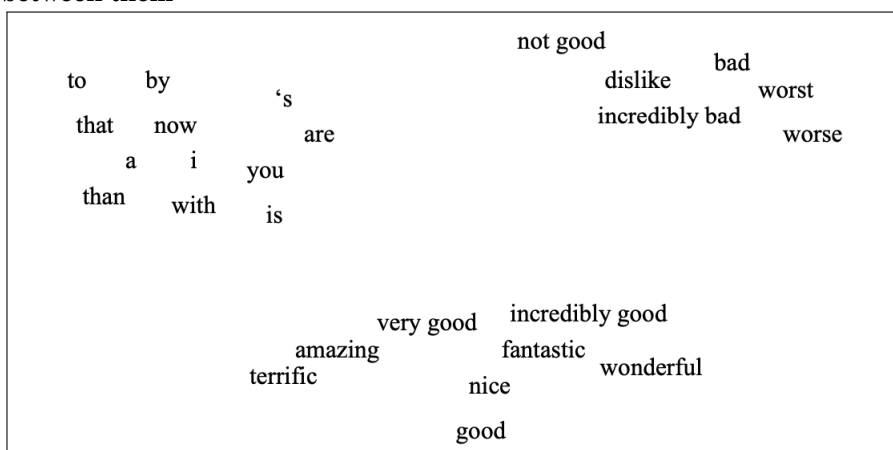
- we'll use an algorithm called SVD from package surprise
- Evaluation and Data Splitting
 - though there's no notion of "accurate" recommendations, we need a way to evaluate our predictions to compare between methods
 - we'll split the data and evaluate our predictions as follows: split the ratings into train and validation sets (not split the utility matrix) **then** create utility matrix for train and validation splits

- during training we assume that we do not have access to some of the available ratings. We predict these ratings and evaluate them against ratings in the validation set.
- Important notes:
 - training matrix is of shape N by M but only has ratings from X_train and all other ratings missing
 - validation matrix is also of shape N by M but only has ratings from X_valid and all other ratings missing
- we can calculate errors between actual ratings and predicted ratings with any metrics of our choice (common is MSE and RMSE)
- baselines: there are a couple of baseline approaches
 - global average baselines: predict all missing value as the global average rating
 - k-NN imputation: impute missing values using mean value from k nearest neighbours found in the dataset
- to see how to actually do collaborative filtering, see "Collaborative filtering → Rating prediction using the surprise package"
- Cross Validation: we can carry out cross validation and grid search using surprise package
- Content Based Filtering:
 - supervised machine learning approach
 - in collaborative filtering we assumed that we only have ratings data, but usually there is some information on items and users available.
 - examples
 - Netflix can describe movies as action, romance, comedy, documentaries.
 - Amazon could describe books according to topics: math, languages, history.
 - we can use these information to make a utility matrix

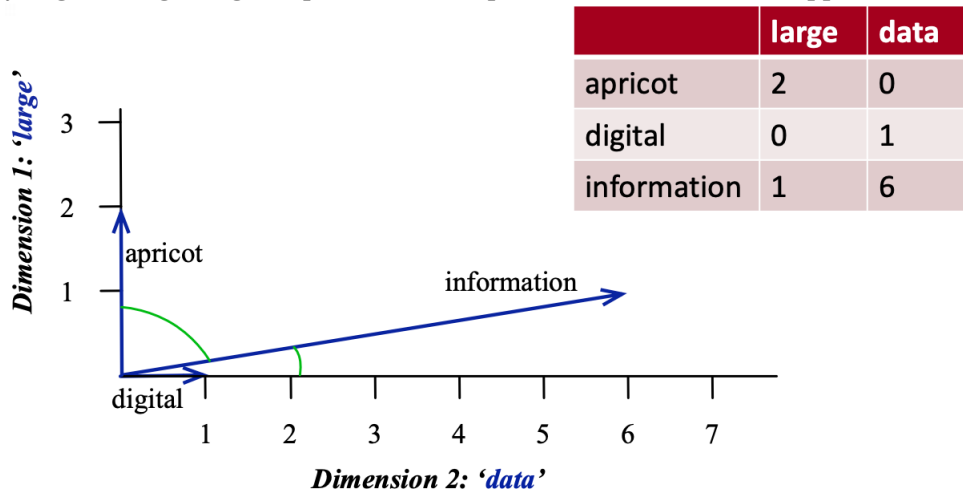
Chapter 16: NLP

Word Embeddings

- word embeddings (representation): idea is to represent word meaning so that similar words are close together
 - so far we've been talking about sentence/document representation (BOW)
 - though word representation can't be used in text classification tasks like sentiment analysis using traditional ML models, they are useful for advanced ML models like neural networks
 - word embeddings allow use to solve meaning-related problems (i.e what is the meaning of "gloves") or which find relationships between words (which words are similar, which ones have positive/negative connotation)
- how to represent words
 - we will need a representation that captures relationships between words
 - we'll be looking at 2 such representations
 - sparse representation with **term-term co-occurrence matrix**
 - dense representation with **Word2Vec**
 - both are based on **distributional hypothesis** and **vector space model**
- Vector Space model: model meaning of a word by placing it into a vector space
 - idea is to create embeddings of words so that distances among words in the vector space indicate relationship between them



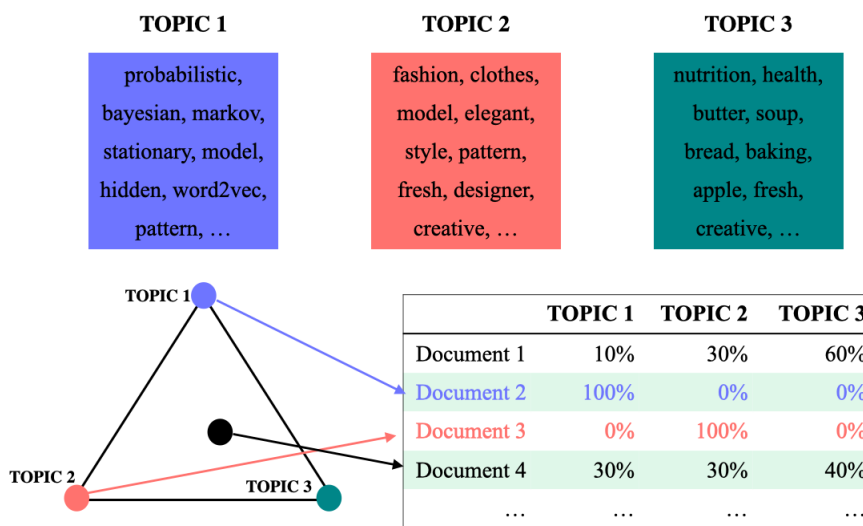
- Term-Term Co-Occurrence Model
 - same idea as bag of words but for texts appearances instead
 - you go through large corpus of text, keep count of all words that appear in context of each word (within a window)



- you can plot these (large column on the y-axis, and data column on the x-axis, with the words being vectors so apricot would be vector $\langle 2, 0 \rangle$)
- similarity within words can be calculated using the dot products between vectors (i.e $\text{digital} \cdot \text{information} = (0 \times 1) + (1 \times 6) = 6 \rightarrow$ higher the dot product the more similar the words)
- similarity can also be captured via cosine similarity
- these matrices are usually are long and sparse (most elements are 0)
 - \rightarrow the alternative is to learn short and dense vectors, these are easier to train with ML models and may generalize better
 - \rightarrow in practice dense vectors they work much better
- Word2Vec: a family of algorithms to create dense word embeddings
 - create dense representation by using gensim or spaCy
 - you can additionally download pre-trained embeddings that's been trained on huge corpus (like Wikipedia)
 - with this you can find words most similar to a word of your choice, or find similarity within words, even find analogies (Lecture 16 \rightarrow word representations \rightarrow finding similar words)
 - Note: there maybe gender/racial stereotypes embedded into the word embeddings
- representing documents using word embeddings: how to represent meaning of paragraphs or documents (assuming we have reasonable representation of words)
 - averaging embeddings
 - concatenating embeddings
- average embeddings: get embedding of each word in the sentence and divide by the number of words
 - ex. all empty promises = $[\text{embedding}(\text{all}) + \text{embedding}(\text{empty}) + \text{embedding}(\text{promise})]/3$
 - using this, you're able to find similarity between documents as well
 - sentiment analysis using average embeddings (as opposed to BOW) reduced overfitting
- Sentence Transformer: fancier methods for document representation
 - better score than both BOW and average embedding but much much slower

Topic Modelling

- suppose you have a large collection of documents on a variety of topics - you want to categorize them so it's easy to search - topic modelling gives you ability to summarize major themes in a large collection of documents
- topic modelling
 - usually solve via unsupervised ML methods
 - give hyperparameter K, describe the data using K topics
 - input:
 - \rightarrow large collection of documents
 - \rightarrow value for the hyperparameter k
 - output:
 - \rightarrow topic-words association: for each topic, what words describe the topic
 - \rightarrow document-topics association: for each document, what topics are expressed by the document



- we'll be training using an LDA model
- process
 1. preprocess your corpus
 2. train LDA using gensim
 3. interpret your topics

Basic Text Preprocessing

- we need to do preprocessing because test data is unstructured and messy
- tokenization
 - sentence segmentation: split text into sentences
 - word tokenization: split sentences into words
- Sentence segmentation
 - you can tell python to do segmentation on punctuation (like periods or exclamation marks)
 - even then, it might be ambiguous (at least period is in English - could be Dr. or U.S or decimal)
 - common way is too use off-the-shelf models for sentence segmentation
 - using nltk
- Word tokenization
 - what are the boundaries we decide for something to be a word (is white space enough) - the process of identifying word boundaries is known as tokenization
 - also use off-the-shelf ML models
 - also use nltk
- types and tokens: types are all **unique** words, tokens are just word counts
- punctuation and stopword removal: removing words like "the", "is", "a" and punctuations
 - also can be done with nltk
- Lemmatization: sometimes we want to ignore morphological differences between words
 - ex. if your search term is "studying for ML quiz" you might want to include pages containing "tips to study for an ML quiz" or "here is how I studied for my ML quiz"
 - can be done via nltk
- Stemming: chops of affixes
 - ex.automates, automatic, automation all reduced to automat
 - can be done via nltk
- every text processing method we said above can be done in spaCy

Chapter 17: Multi-class classification and Computer Vision

Multi-class Classification

- sometimes we need to do classification with more than 2 classes
- many linear classification models don't extend naturally to the multi-class case, so we'll need some techniques to get around this
- 2 kinds of approaches
 - one-vs-rest approach (OVR)
 - one-vs-one approach (OVO)
- One vs. Rest
 - learn a binary model for each class which tries to separate that class from all other class
 - ex. you have 3 class, you'd try out:
 - 1 vs {2,3} (so either class 1 or the other)
 - 2 vs {1,3} (either class 2 or the other)
 - 3 vs {1, 2} (either class 3 or the other)

- if you have k classes, it'll train k binary classifiers, one for each class
- it'll be train on imbalanced datasets containing all examples
- given test point, get scores from all binary classifiers (e.g. raw score for LR) → the classifier which has the highest score, choose the one class from that classifier
→ ex. given test point, $LR_1 \text{ vs } \{2,3\} = 0.6$, $LR_2 \text{ vs } \{1,3\} = 0.5$, $LR_3 \text{ vs } \{1,2\} = 0.3$, predict class 1
- since we have k classifier, we will have coefficients and intercept for each of those classifiers (enough to make k lines)
→ you can also get k decision boundaries which makes it a little easier to predict new points
- **One vs. One Approach**
 - build a binary model for each pair of classes
→ ex. 1 vs 2, 1 vs 3, 2 vs 3
 - will train $\frac{n(n-1)}{2}$ binary classifiers
 - trained on relatively balanced subsets (will ignore the other classes - for example, for the 1 vs 2 classifier, it will ignore all points of class 3)
 - predictions: apply all classifiers on test example → count how often each class was predicted → predict the class with the most votes
- OVO takes more time than OVR
- unlikely you will need to use them because most models have multi-class support

Intro to Computer Vision

- computer vision: understanding images/videos, esp using ML/AI (neural network specifically in our case)
- neural network: apply a sequence of transformations on your input data
 - involves a series of transformations (layers)
 - think about it a bit like LR - doing weighted sums, but what makes neural network more powerful than linear models is that we can apply non-linear function to weighted sum for each hidden nodes
- pros of neural network:
 - can learn very complex functions (more/bigger layers = more complex model), you can generally get a model that will not underfit
 - works well for structured data (i.e 1D sequence - like time series, language; 2D image; 3D image/video)
 - transfer learning (later) is very useful
- cons of neural network:
 - require a lot of data
 - computationally expensive
 - have a lot of hyperparameters which are hard to tune
 - not very interpretable
 - when you call fit, you are not guaranteed to get the optimal
- we can try to use basic ML like LR on tabular image data (squashed into array of numbers that represent numbers - "flattening the image") but our results are garbage
 - flattening the image throws away useful information
- transfer learning: use pre-trained (neural network) models (like convolution neural networks - CNN) and fine tune to your need

Chapter 18: Time Series

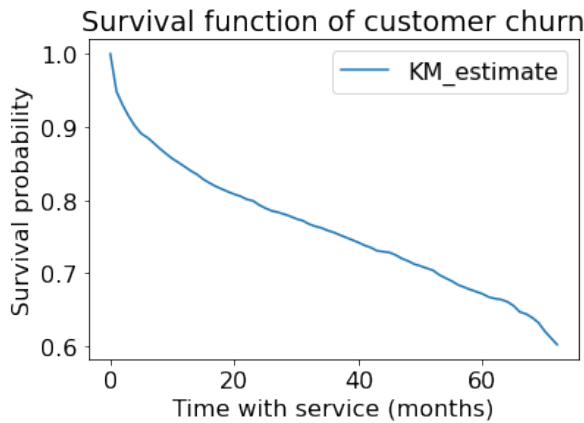
- time series is collection of data points indexed in time order
 - notice when datasets have datetime features
- questions we're looking to answer: how many people likely to rent a bike at this station tomorrow at 3pm given everything we know about rentals in the past
- need special way to split time series data
 - we don't want to split normally - has chance that we are training on data that came after our "test data" → so if we want to forecast, we aren't allowed to know what happened in the future
 - what we do is treat data before a certain date as training data and everything after as test data
→ ex. if we have total 248 data points, we'd use the first 184 data (corresponds to first 23 days in a month) as training data and the remaining 64 for the last 8 days as test data
- training models
 - how to encode time features: using POSIX time
 - our prediction test (question above) is a regression task
 - note: be careful when working with datasets with **only** time features
→ **tree-based models cannot extrapolate to feature ranges outside the training data** (see lecture 18 → training models)
- feature engineering for date/time columns:
 - if our index is of the special type DateTimeIndex - we can extract a lot of interesting from it, like:
 - get month names for each example
 - get days of the week
 - get day of the week

- hours of the day
- just by using time of the day as our feature column we can get better results
- we can get even better results by adding a day of week feature as well
- trying Ridge on data
 - Ridge performs a bit poorly on both training and testing data (not able to pick up periodic pattern)
 - reason is because we encoded time of day as integers and linear functions and only learn a linear function of the time of day
 - turning hour and day feature into categorical values really helps
 - even better if you apply PolynomialFeatures transformer
 - looking at coefficients learned by Ridge useful to predict what time is the most busy
- Cross Validation:
 - again, problem with splitting training data into training and validation set
 - use TimeSeriesSplit for time series data
- can turn string Dates columns into DateTimeIndex by using pandas' to_datetime() function
- pretty intuitive example using Australia rainfall in Lecture 18
- lag-based features:
 - sometimes there's temporal dependence: observations closer in time tend to be correlated
 - ex. what if tomorrow's rain fall is related to yesterday's features, or the day before
 - this is particularly when the change from one week to another isn't exactly drastic (i.e if next week's avocado price is slightly different from this week then lagged features are a good idea, however, if they fluctuate a lot then it's not a good idea)
 - this is called a lagged (or shift feature)
 - use pandas' shift function → basically add a lagged feature columns (i.e lagged rainfall feature for day 2 is rainfall feature from day 1)
- forecasting further into the future (i.e predict rainfall 7 days into the future)
 1. train a model for each of the days (one for tomorrow, one for the day after, etc) and we can build these datasets (because we have historical data - of course, as the days into the future gets bigger, the dataset gets smaller)
 2. use multi-output model (not in scope of class)
 3. use one model and sequentially predict using for loop (use our predictions as truths and iterate through)
- trends: rely on having a continuous target
 - in our example, we used only lag features (ex. sales, sales-1, sales-3, sales 4,)
 - if you have a DaySince feature and use Linear Regression on it, the coefficient learned will predict unlimited growth forever if it's positive → this is your trend
 - if you use Random Forest, we'll be doing splits from the training sets (i.e if DaySince > 9100, do this) and thus there will be no splits for later time points because there's no training data
 - **tree-based models cannot model trend**

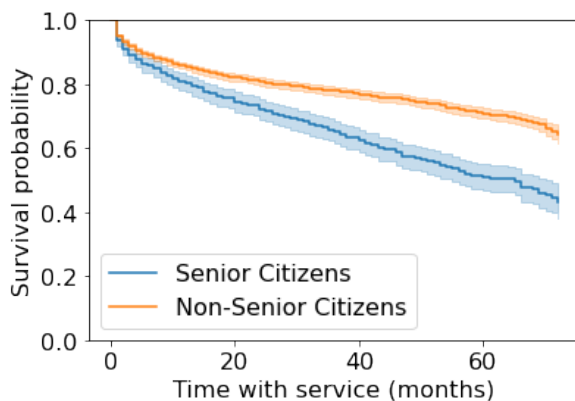
Chapter 19: Survival Analysis

- useful for problems like the "customer churn" datasets and incorporating the time feature
- when we did binary classification (churn vs not churn), we're missing out on the time aspect - we can't say how long a customer will stay, or how likely they'll stay based on how long they've been with the company
- time to event
 - sometimes you want to analyze the time till an event happens (ex. time until a piece of equipment breaks)
 - so in our churn, instead of predicting churn or not churn, it's more useful to when the customer is likely to churn
- censoring: when a patient is censored we don't know the true survival time for that patient
 - ex. if we're predicting a tenure of a customer, but we don't know the actual tenure times for tuples where churn = no (because, well, their tenure hasn't ended yet → this is right-censoring)
 - this is a problem → means that we don't have correct target values to train/test model
 - approach to solve
 1. only consider examples where churn = "yes"
 - on average, our predicted tenure time will be underestimates because we are ignoring currently subscribed (unchurned customers)
 - our dataset is a biased samples of those who churned during data collection time
 2. assume everyone churns right now
 - in other words, use the original dataset
 - our prediction will again be underestimates, because for those still subscribed, we recorded a shorter total tenure time than reality - because they will keep going for some amount of time
 - survival analysis: the proper way to deal with this
- survival analysis: use the lifelines package

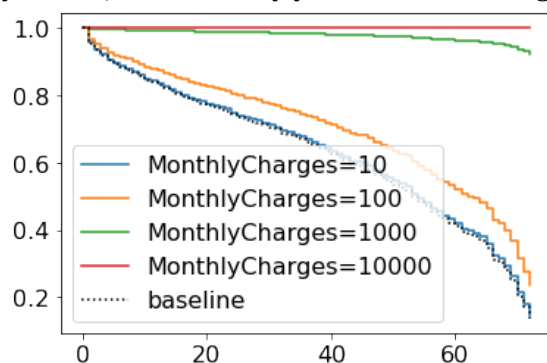
- Kaplan-Meier survival curve: part of the lifelines package
 - for the model, in our example, we only use 2 features - tenure and churn
 - we get this curve



- the plot is telling the probability of survival over time
 - ex. after 20 months, the probability of survival is about 0.8
 - steep drop in the beginning that people tend to leave early on
- you can create K-M curve for different subgroups (i.e seniors citizens) and we can see that senior citizens tends to churn more quickly than others



- Cox proportional hazards model
 - incorporates other features into the model, produces similar survival curve
 - Cox proportional hazards model is a commonly used model that allows us to interpret how features influence a censored tenure/duration (a bit like linear regression for survival analysis - we'll get coefficient for each feature but it'll tell us how it influences survival)
 - blah blah we run it on our model, get the coefficients and $\text{Contract_Month-to-Month} = 0.812874$ and $\text{Contract_Two year} = -0.776425$ so this means that month-to-month leads to more churn while two-year contracts leads to less churn → so negative is actually good for companies
 - **Cox proportional hazards model assumes the effect of a feature is the same for all customers and over all time**
 - we can get the survival curve as well, let's look at the different Monthly Charges, from this, we see that the bigger your bill, the less likely you are to churn (higher survival rate) so we should expect a negative coefficient

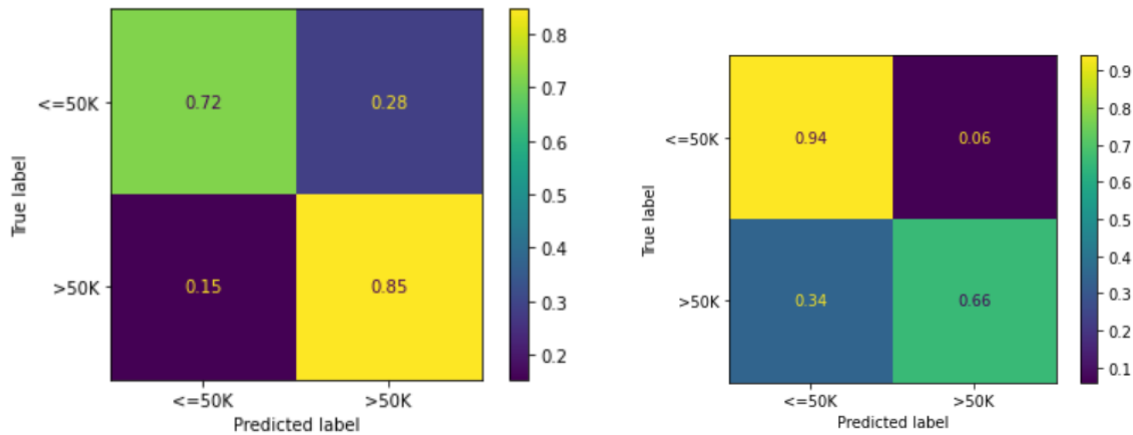


- predictions: we can use survival analysis to make predictions
 - we can predict how long each non-churned customer is likely to stay according to the model assuming that they just joined right now (so we drop the churn column)
 - you can condition the function on certain things (i.e how long they're gonna stay given they've been here 20 months - instead of assuming starting now)
 - thus, we can then do a graph of how long each non-churned customer is likely to stay according to the model assuming they've been here for the tenure time

- evaluations: by default returns "partial log-likelihood"
 - however, concordance index (c-index) is more interpretable
 - 0.5 is expected results from random predictions
 - 1.0 is perfect concordance
 - 0.0 perfect anti-concordance (multiply results with -1 to get 1 - so predicting opposite)

Chapter 20: Ethics

- here, we use the fraud banks example and observe the confusion matrix between the two (it's male only on the left and female only on the right)



- notice that there are more false negative for females, so even though they make more than 50k, the model assumes otherwise
- also notice that there are more false positives for males
- we see that accuracy male = 0.756 while accuracy female = 0.910 → could be explained by class imbalance
- if we do value counts we see that there's a bigger class imbalance for females [0.90, 0.10] vs males [0.70, 0.30]
- consequence: if you're just looking at predicted income to decide to approve loans or not, male has more chance of getting loans approved
- statistical parity suggests that the proportion of each segment of a protected class (e.g. sex) should receive the positive outcome at equal rates
 - ex. number of loans approved for female should be equal to male
- equal opportunity suggests that each group should get positive outcomes at equal rates (assuming they qualify for it)
 - true positive rate (TPR or recall) of both groups should be equal → should look at recall
 - we see that recall male = 0.847 and recall female = 0.657
 - so we can see that we're not giving equal opportunity to men and women (men gets approve more)
 - vastly different recall scores indicate that we're not giving equal opportunity to both groups
- moreover, banks usually want to approve as many qualified people as possible (true positive), but also to avoid approving unqualified application (false positive) so we should look at false positive rate (FPR)
 - we get FPR male = 0.285 and FPR female = 0.060
 - so we're getting a lot more false positives for men, so we should also take a look at FPR between groups

Chapter 21: Communications

- we want to be confident when we're right, but hesitant when we're wrong
 - use credence (the bolded parts below)
 - if Toronto has the highest cost of living and our model predicted otherwise, we want to express hesitancy
 - ex. Vancouver has the highest cost of living in Canada. **I am 55% sure of this**
- loss functions: when you call fit for LogisticRegression it has these same preferences

correct and confident > correct and hesitant > incorrect and hesitant > incorrect and confident

- sklearn models use a log loss function → it's an "error" function, so lower values are better
- fit tries to minimize this error
- so **in terms of log loss score**

$loss(\text{correct and confident}) < loss(\text{correct and hesitant}) < loss(\text{incorrect and hesitant}) < loss(\text{incorrect and confident})$