SUMMARY OF WORLDQUANT MODULE 2

SECTION 1 – INTRODUCTION TO MACHINE LEARNING AND SCIKIT-LEARN API (in.ipynb)

UNIT: Introduction to Machine Learning (ML_Intro_ML.ipynb)

- ML finds pattern in data
- The availability of more data is now and improvement in computer memory and performance have caused major improvement in ML
- Supervised and Unsupervised Learning
- Unsupervised learning used for clustering, dimensionality reduction
- Rows correspond to the "observation" while columns correspond to the "characteristics/attributes/degree of freedom"
- Try to make our model capture from the "signal" and not the "noise"
- Somewhere between "under fitting" and "overfitting" lies our "best model"
- Hyper parameters: they are set prior to fitting of the model, a level above normal/simple parameters, they govern the learning process
- Scikit-learn: python main package for Machine Learning, offers ML models to use and other useful stuff
- Most ML models are Classes
- "Classname?" > to get the documentation
- "y = m*X + b": model for linear regression
- This course will talk about working with texts

UNIT 1.2.1: Intro to Scikit-learn (ML_Scikit_Learn.ipynb)

- Scikit-learn main python package for ML
- Sklearn provides ML models, provides ready-made datasets and other needed tools (so we do not reinvent the wheel)
- Google "online whiteboard"
- Sklearn uses a lot of OOP principles
- Custom classes names start with a "Capital letter" uses "Camel Cases"
- In sklearn ML models are referred to as "Estimators" (types: classifier, regressors (predictors), transformers)
- Create our own custom estimators…(interesting)

UNIT 1.2.2: Predictors (ML_Scikit_Learn.ipynb)

- Supervised learning is either classification or regression
- Two main methods of Predictors are: ".fit(X, y)" and ".predict(X)"
- Make it a habit to look up Documentations

- ".score(X, y)": to see how well our model perform, "R2" for regression and "accuracy" for classification

UNIT 1.2.3: Transformers and Pipelines (ML_Scikit_Learn.ipynb)

- Transformers: carry out a form of transformation on our data
- ".fit(X), .transform(X), .fit_transform(X)" > Interface for Transformers
- "StandardScaler()": a transformer that is often used (scales from -1 to 1)
- "Column Transformers()": when we want to transform specific columns
- Use "Pipelines" to aggregate steps together to make things easier, "transformers" come before the "predictor" in the Pipeline
- "pipe.named_steps"
- Pipeline objects are also "estimators"

UNIT 1.2.4: Feature Unions (ML_Scikit_Learn.ipynb)

- Feature Union: it holds transformers in parallel, then combine them in the end
- The order of steps in feature union does not matter since they run in parallel

UNIT 1.2.5: Custom Transformers (ML_Scikit_Learn.ipynb)

- Encouraged to check out the Sklearn package and all it offers
- Creating "Custom Estimators" to cater for very specific needs
- Sklearn relies on OOP principles
- All estimators inherit from the "BaseEstimator"
- "TransformerMixin" makes it a transformer estimator
- Axis = 0 > across the rows, Axis = 1 > across the columns

UNIT 1.2.6: Custom Predictors (ML_Scikit_Learn.ipynb)

- "RegressorMixin": defines the estimator as a Regressor

UNIT 1.2.7: Exercise Distance Transformer (ML_Scikit_Learn.ipynb)

- It helps to sketch out our workflow before we implement it

UNIT 1.2.8: Exercise Majority Classifier (ML_Scikit_Learn.ipynb)

UNIT 1.3.1: Persisting Your Model (ML_Scikit_Learn.ipynb)

- Persist our model == Save it to disk > that is to just predict with the model and not have to retrain it every time
- "import dill": similar to python pickle object
- You can use "dill" with "gzip" to compress large models

UNIT 1.3.2: Common Mistakes ()

- Advantages of Pipeline

SECTION 2 – REGRESSION, CLASSIFICATION & MODEL SELECTION (ml.ipynb)

UNIT 2.1.1: Regression Metrics (ML_Metrics.ipynb)

- Supervised ML have labels/targets for training
- Two Branches of supervised learning ML: Regression (continuous value) and Classification (discrete values)
- Different metrics for regression and classification
- Mean Squared Error (MSE) > penalizes error a lot
- Mean Absolute Error (MAE) >
- You want the error to be as small as possible
- A metric that compares out model to a bench mark model we use "R^2 " > Coefficient of determination, compares your MSE to your benchmark's MSE
- "R^2 " < 0 means our model is bad, does worst that the benchmark
- "R^2 " does not change with the scale of our label, it is invariant to scale of label

UNIT 2.2.1: Linear Regression Intro (ML_LinearRegression.ipynb)

- Linear regression assumes there is a linear relationship between data and label/target
- It is not a complicated model
- We always try to minimize the "cost function"
- Read documentations!!!!!
- "linear.coerf_" > outputs the gradient for the linear regression

UNIT 2.2.2: Gradient Descent and Huber Loss (ML_LinearRegression.ipynb)

- One way to determine the best parameters (minimize cost function) is by using "Gradient Descent"
- Picture it as if you are descending to a valley
- The value of our Gradient Descent steps determine if we find the optimal value for minimum cost function
- HuberRegressor: minimizes a different "cost function" from that of the LinearRegression
- HuberRegressor will better handle "Outliers" with its modified approach

UNIT 2.2.3: Multivariate Regression (ML_LinearRegression.ipynb)

- Checks out the plots with widgets in the notebook

UNIT 2.2.4: Feature Importance (ML_LinearRegression.ipynb)

- Scale the features to allow easier comparison between them
- Linear Regression is a simple model, easy to derive insight from
- Simple Model sometimes serve as benchmark (I use null models most time)

UNIT 2.3.1: Classification Metrics (ML_Classification.ipynb)

- Branch of supervised ML
- Multiclass Classification: more than 2 discrete labels
- Accuracy: number of correct observation/ number of observations
- We have consider the balance between the labels, it helps to choose the metric to use for evaluation
- Precison: TP/ All positive prediction (TP + FP)
- Recall : TP/ all positive observations (TP + FN)
- For disease detection higher RECALL is better, sometimes higher PRECISION is better, there is a trade-off between them. We might have to sacrifice one for the other
- Different values for precision and recall for whatever label is classified as “Positive”
- “metrics.classfication_report()”
- “f1_score”: is a harmonic mean of “precision” and “recall”

UNIT 2.3.2: Probabilistic Models and Metrics (ML_Classification.ipynb)

- “Log Loss” (Cross Entropy): takes into account the “uncertainty” of your model, which accuracy does not do
- “Probabilistic Model”: Classification Models that return probabilities not a label, they have “thresholds” that can be modified which affects the “precision” and “recall”
- “AUC”: measures the “precision”, “recall” tradeoff, the higher the AUC the lesser the trade off
- “ROC-AUC”: similar to AUC, plots the TP rate against the FN rate
- In general, If your data is imbalanced you should rely on AUC

UNIT 2.3.3: Logistic Regression (ML_Classification.ipynb)

- Logistic Regression is the classifier version of linear regression, it is a probabilistic model, uses a “sigmoid” function (output will always be between 0 and 1)
- “log loss”: error metric (cost function) used by probabilistic models
- “decision boundary”: tells us how the model makes predictions (boundary between decisions the model wants to make)
- Tries to draw a line (hyperplane) that best separate the classes by minimizing the “log loss”

UNIT 2.3.4: Multi Classification (ML_Classification.ipynb)

- Logistic Regression has a deficiency “it is a binary classifier”
- Two common schemes for modifying “binary classifier” to handle multi-class labels: 1) One vs All 2) One vs One
- One vs All: we train K (no of classes) classifiers, reweigh the probabilities, each K classifier gets trained on the whole dataset
- One vs One: train a classifier for each pair classes, they are trained for a subset of the dataset

- Sklearn allows you to specify the scheme you want to use for your model

UNIT 2.4.1: Model Selection (ML_ModelSelection.ipynb)

- Check out the Sklearn Algorithm Cheat-Sheet
- Check if you have more than 50 samples
- We use “Unsupervised ML” when our data does not have a “label”
- Initial exploration (EDA) with the features we have

UNIT 2.4.2: Intro to Decision Trees (ML_ModelSelection.ipynb)

- Decision Tree has a “non-linear model”
- They use a binary tree model, like a flowchart
- “max-depth”: “hyper-parameter” that governs how far the tree grows
- It defines the observations into “bins”, and predicts the value of the “bins”
- Some ML Model are based on Decision Tree
- You can easily visualize/explain how this model works
- Play around with the hyper-parameters to make Models perform better

UNIT 2.4.3: Under-fitting and Overfitting (ML_ModelSelection.ipynb)

- There is always going to be noise in our data
- “if we memorize, we are not really learning anything”
- We want our Model to “learning” and not “memorize”
- The best “Hyper-parameter values (max-depth)” (where the model does not learn from the “noise”) will vary different dataset/ problems
- The data split usually between 10-30%
- “random_state”: helps to maintain reproductivity
- Always check out documentations
- Always want to find the sweet spot between “Underfitting” and “Overfitting”

UNIT 2.4.4: GridSearchCV (ML_ModelSelection.ipynb)

- “Hyper Parameter Tuning”: process of finding the optimal values for model hyper paramters
- It is important to know the “Hyper parameters” our model has
- “You don’t always have to reinvent the wheel” (in regard to custom classes)
- “GridSearchCV (model, {dictionary of parameters to tune}, cv = no of folds, n_jobs, scoring, verbose)”, its constructs a “n-dimensional grid”
- More folds increase “training time” but reduce “randomness”
- “n_jobs = -1”: uses all the processors of your system
- “verbose = 1”: tells us what is going on as it happens
- “gs.best_params_”, “gs.best_estimators_”

UNIT 2.4.5: Comparing Two Models (ML_ModelSelection.ipynb)

- You can only tune one model at a time with GridSearchCV
- Check out the code here…its sleek
- Sometimes making predictions is not your only priority, your choice of model is based on your priority (faster training time, better predictions and so on)

UNIT 2.5.1: Imputation ()

- Dealing with missing data: 1) remove entries with missing values, it might affect our model performance 2) replace the missing value
- Usually one of the first steps in preprocessing
- We usually use "mean/median values" for numerical imputations
- "Imputer/SimpleImputer()": sklearn transformer for "imputation"
- In general a "median" is a better strategy because the "mean" is susceptible to outliers

UNIT 2.5.2: Categorical Data ()

- Our Supervised ML models works with numbers (structured data)
- ".unique()": returns the unique values of the columns
- Try to learn how to use "map()" function
- "LabelEncoder()", ".classes_": gives the mapping of the values
- Categories don't always have a natural order to them, so we sometimes use "One Hot Encoder"
- Good to use "pipelines" for preprocessing
- ".toarray()": to make "sparse matrix" readable (I think)

UNIT 2.6.1: GridSearchCV and Pipelines 1 ()

- Using of GridSearch effectively with Pipelines requires a bit of practice
- Pipeline and GridSearchCV are Estimators
- "Pipeline.get__params()"
- It is good to use "np.logspace()" to generate parameters for "alpha"
- "from tempfile import mkdtemp" > "from shutil import rmtree": help to speed up the GridSearchCv by storing saved/run iterations on disk

UNIT 2.6.2: GridSearchCV and Pipelines 2 ()

UNIT 2.6.3: RandomizedSearchCV ()

- Computational cost of Gridsearch increases exponentially with increase in parameters and datasets to search through
- Randomized Search CV: works like Grid by creating the candidate but selecting a random number of the candidates not all the candidates. It is computational less costly, but does slightly worse than GridSearch
- RandomizedSearchCV(n_iter = (how many samples should be sampled))

SECTION 3 – FEATURE ENGINEERING

UNIT 3.1.1: Feature Engineering and Extraction (ML_FeatureEngineering.ipynb)

- If there is no structure/pattern in our data then the model will not reveal anything
- "df.corr()": to check out the correlations
- Feature generation/engineering is an "open ended task"
- Text data and image data are not in "structured format"

UNIT 3.1.2: Feature Transformation (ML_FeatureEngineering.ipynb)

- Remember simple models are easy to work with and get insight from
- Using "np.exp()" to transform data, inorder to help out linear models

UNIT 3.1.3: Curse of Dimensionality (ML_FeatureEngineering.ipynb)

- Sometimes adding more data improves ML model performance
- We increase "dimensionality" by adding more features
- "curse of dimensionality": at a point the feature become too much and the model starts to perform poorly if we don't add more observations
- "RS.best_estimator_", "RS.best_estimator_.feature_importance_"
- "list(zip())"
- Collecting data is an expensive process, easier said than done
- Lot of feature selection tools in SKLEARN
- "from sklearn.feature_selection import RFECV": a recursive feature selection

UNIT 3.1.4: Regularization (ML_FeatureEngineering.ipynb)

- Adding too many features can be a problem, we don't want our model to be adaptive to "noise"
- Regularization help to prevent overfitting, a regularization parameter is added to our cost function (real life example: regularization of the commodity market)
- Regularization reduces the impact of the beta coefficient
- Reduce Reg parameter (alpha) to handle "underfitting" and increase Reg parameter to handle "overfitting"
- When doing regularization you need to scale your model in order to put the features on the same scale
- Lasso: can push beta coefficients down to zero (0) making then insignificant

UNIT 3.1.5: Multicollinearity and PCA (ML_FeatureEngineering.ipynb)

- Highly correlated features bring about instability in our linear regression models
- Start exploring your dataset first, checking the correlations "df.corr()"
- Correlation can be an indicator of redundancy

- PCA: it is a transformer, helps to reduce multicollinearity, helps to generate non-correlating features without losing much information, it projects the data into a lower dimensional space

UNIT 3.1.6: Ensemble Models ((ML_FeatureEngineering.ipynb)

- Ensemble models: use more than one estimator in its decision making
- Random Forest: ensemble of decision tree ("wisdom in the crowd")
- We can create out custom ensembles
- Blending ensemble (assumes one model prediction might have more weight than the others been ensemble), unlike Random Forest where we bag the result of the models
- We can use linear regression (we might set intercept = False) (or any other model) to blend ensembles
- "np.atleast_2d": to convert to 2D array
- ".T": to do a transform

UNIT 3.2.1: Bias and Variance ()

- Error can have 3 components: Bias, Variance, Irreducible error (mostly from data collection)
- Underfitting – High Bias (simple notion of something)
- Overfitting – High Variance, the more complex the model its more susceptible to overfitting
- A model will generally do better when predicting from the training data than the test data
- Difference in the training score and test score gives us an idea of the level of variance
- No need to add more data to a model of high bias
- Adding more data will help a model of high variance
- Train error will always be less than the test error
- Difference between the test error and train error tells us if more data will be needed or helpful

UNIT 3.2.2: Learning Curves

- Screenshot of learning curve

UNIT 3.3.1: Intro KNN (ML_K_Nearest_Neighbors.ipynb)

- KNN is a simple machine learning algorithm, it is like an interpolation technique
- KNN can easily handle multiclassifcation
- In KNN we have discern a "distance matrix"

UNIT 3.3.2: KNN Bias and Variance (ML_K_Nearest_Neighbors.ipynb)

- We use "Cosine Similarity" if we are using the direction of the points not the distance between them
- "gs.cv_results_", inspect for the best parameters

UNIT 3.3.3: KNN Time Complexity (ML_K_Nearest_Neighbors.ipynb)

- KNN is not considered for large dataset
- Other algorithms learn from the training data by generating a set of parameters
- For fast predictions stay away from KNN

UNIT 3.3.4: KD Trees and Weights ()

- KD Tree helps us to reduce the time complexity of KNN
- "algorithm": KNN hyper parameter for choosing KD Tree or not
- "weight"
- Hyper parameter don't have to be integers alone

UNIT 3.4.1: Intro to NLP (ML_Natural_Language_Processing.ipynb)

- NLP is a field devoted to methods and algorithms for processing human (natural) languages. NLP is a vast discipline
- Text data is unstructured, so a lot of preprocessing is needed
- ML applications of NLP: sentiment analysis, topic modelling and language translation
- Corpus: the body/collection of text being investigated
- Document: a single unit of analysis, a single observation

UNIT 3.4.2: Spacy (ML_Natural_Language_Processing.ipynb)

- Spacy a Python package for NLP
- Spacy provides support for other languages: spacy.load("en")
- "word.pos_": returns the part of speech
- "word.tag_": more specific the pos
- "spacy.explain(tag)"
- Tag meanings: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html
- Spacy is powerful python package for NLP, check out the Spacy website/documentation

UNIT 3.4.3: Obtaining a Corpus (ML_Natural_Language_Processing.ipynb)

- There is a python package called "wikipedia", makes it easier to fetch articles
- Choosing a unit of observation (document) we have to weight the pros and cons

UNIT 3.4.4: Bag of Words Model (ML_Natural_Language_Processing.ipynb)

- Machine learning need to take in structured data (rows and columns) and also in number format
- Bag of words counts the frequency of appearance of each word in our sentence
- "CountVectorizer": a bag of words model found in Sklearn
- "Vectorizer": refers to a transformer that converts a data structure (like a dictionary) into a Numpy array

- Bag of Words model is a simple model, it disregards word order, context and grammar, but it gets the job done
- “min_df”, “max_df”, “max_features”: key hyper parameters
- “.get_feature_names()”: returns a list of words used as features
- “.vocabulary_”: unique words together with their column/index
- You cannot invert transform with the Bag of Words model because when it transforms it loses the structure of the words

UNIT 3.4.5: Hashing Vectorizer (ML_Natural_Language_Processing.ipynb)

- Hash Vectorizer: helps to improve on the time complexity of countvectorizer by paralleizing the process, it returns the same output for a specific input
- Hash Vectorizer is good for “online learning”

UNIT 3.4.6: TF-IDF (ML_Natural_Language_Processing.ipynb)

- Both Count Vectorizer and Hash Vectorizer create a feature of raw counts only
- Term Frequency-Inverse Document Frequency – TF-IDF
- With TF-IDF words that are common throughout the documents get weighted down and vice-versa
- With TF-IDF weighting scheme, a ML model will have an easier time to learn patterns
- “Tf-idf Vectorizer”: combines both “Count Vectorizer/ Hash Vectorizer” with “Tf-idf Transformer”
- Tf-idf Vectorizer works on an array of documents, while Tf-idf Transformer works on a count matrix
- Tf-idf outputs floats

UNIT 3.4.7: Improving Signal (ML_Natural_Language_Processing.ipynb)

- It is important to create a baseline model
- Spacy has a nice collection of stop words
- Examine the stop words to see that you don’t want to add/remove anything
- “Stop Words”
- “Stemming and Lemmatization”: Stemming returns the “stem” word (not necessarily a dictionary word), while Lemmatization is a more sophisticated process it returns it “lemma” (a dictionary word)
- Lemmatization is not 100% accurate
- “word.lemma_”: return the lemma
- “tokenizer”

UNIT 3.4.8: N-grams and Similarity (ML_Natural_Language_Processing.ipynb)

- Tokens refer to what we are counting
- "ngram_range(min_n, max_n)"
- As we increase our n_grams we increase our dataset
- People rarely go beyond bi-grams
- We should generally implement stop words
- Document similarity: what are similar documents in our corpus, we can use the cosine similarity (angle between them), can also use the "dot product"

UNIT 3.4.9: Word Usage Classifier

- "Bi_grams, stop_words, lemmatization"
- Naïve Bayes model is a simple probabilistic model often use as baseline model in NLP
- Hurray our classifiers does a decent job

UNIT 3.4.10: Exercise I

- How much does Stop Words and Lemmatization help?

UNIT 3.4.11: Exercise II

- "gs.cv_results_": returns a dictionary of the summary of the result
- Grid search helps to choose best hyper parameters, it also investigates the effect of each parameter and their runtimes

UNIT 3.4.12: Exercise III and IV

- We can also scrap data from "reddit"
- Using the "nb.coef_" to get the most descriptive word/feature

SECTION 4 – DECISION TREES

UNIT 4.1.1: Intro to Decision Trees (ML_Tree_Based_Models.ipynb)

- Decision Trees are general class ML models (classification and regression), they mimic human decision making process (yes or no)
- Models with low transparency are called "black boxes", it is hard to get insight over the process they are modelling

UNIT 4.1.2: Tree Error Metrics (ML_Tree_Based_Models.ipynb)

- For classification Decision Tree uses the Gini impurity metric
- We can also use “entropy” as our classification metric
- By default, “Decision Tree Classifier” uses the Gini metric, using either of them will not make substantial difference in your classifier

UNIT 4.1.3: Tree Regression (ML_Tree_Based_Models.ipynb)

- For decision tree regression we aim to achieve an overall drop in variance by performing a split
- We are creating a bin, depending on the bin an observation we predict its value

UNIT 4.1.4: Training Trees and Hyper parameters (ML_Tree_Based_Models.ipynb)

- The “node splitting” algorithm is a “greedy algorithm” (it ask the question of now, “what do I do now to drop the error metric”)
- Greedy algorithm has lesser time complexity
- Key hyper parameters: “max_depth”, “max_features”, “min_samples_split”, “min_samples_leaf”
- “max_depth”: too deep and it starts to over fit

UNIT 4.1.5: Geometric Interpretation and Time Complexity (ML_Tree_Based_Models.ipynb)

- Decision boundary helps us to visualize how our model predicts
- There is no need to scale data when using Decision Tree because it does not compare the features, it considers them individually
- Decision Tree might have difficulty with highly correlated features
- We can use PCA
- It is faster to make predictions with Decision Tree than to train it

UNIT 4.1.6: Time Complexity Continued (ML_Tree_Based_Models.ipynb)

- For prediction on Decision Tree increase in data has a slight increase in prediction time
- For training on Decision Tree increase in data has a “more than double” increase in training time
- Check out “Big O complexity cheat sheet”

UNIT 4.1.7: Random Forests (ML_Tree_Based_Models.ipynb)

- Ensemble models are ML models that use more than one predictor to arrive at a prediction
- 3 types of ensemble models: bagging, boosting and blending
- People have used decision tree to construct bagging and boosting based models

- “Aggregation” mimics the idea of “wisdom of the crowd”
- For random forest to be effective the model needs a diverse collection of Trees
- “Bootstrapping”: is a technique to generate new dataset with a single set by random sampling with replacement
- Random forest is a bagging ensemble models, it will take longer to train
- “n_estimators, n_jobs, warm_start”

UNIT 4.1.8: Extreme Random Forests (ML_Tree_Based_Models.ipynb)

- “ExtraTreesClassifier” helps to reduce “overfitting”
- For Random forest, double the feature you double the prediction time, the increase in training time is not so linear since we only consider subset of features
- Extra Trees Classifier also has a faster training time
- Always Check out the documentations

UNIT 4.1.9: Gradient Boosting Trees I (ML_Tree_Based_Models.ipynb)

- Bootstrap + Aggregation = Bagging Ensemble, models are trained in parallel then aggregated together
- “Boosting”: refers to the algorithm’s ability to combine weak leaners to form a strong learner, the models are trained in series

UNIT 4.1.10: Gradient Boosting Trees II (ML_Tree_Based_Models.ipynb)

- Gradient Boosting trees have a similar set of hyper parameters as random forests but with some additions: “learning_rate”, “subsample”
- It is robust to overfitting
- “GradientBoostingRegressor”
- You have to drop the learning rate if you are subsampling
- Decreasing the learning rate requires more trees

UNIT 4.1.11: Feature Importance (ML_Tree_Based_Models.ipynb)

- Tree models have the capability to evaluate feature importance
- “model.feature_importances_”

UNIT 4.1.12: Exercises (ML_Tree_Based_Models.ipynb)

- It is quite tricky to tune hyper parameters of models

SECTION 5 – SVM AND CLUSTERING

UNIT 5.1.1: Intro to SVM (ML_Support_Vector_Machines.ipynb)

- SVM they are best used for small/medium datasets
- SVM can be used for classification, regression and anomaly/outlier detection
- Hard margin classifier is a model that uses a hyperplane to completely separate two classes
- Our models should not only seek for complete separation of the classes but also create the largest margin between the classes

UNIT 5.1.2: Largest Margin Classifier (ML_Support_Vector_Machines.ipynb)

- Support vector that help maintain or support the structure (margin boundaries), the support vectors are the only vectors in the training set that influences the choice of hyperplane
- It is prone to variance error because it is dictated by a few number of training data

UNIT 5.1.3: Soft Margin Classifier (ML_Support_Vector_Machines.ipynb)

- Sometimes it is not possibly to completely separate our points, we adapt/modify our hard margin classifier to a "soft margin classifier"
- There will penalty for margin violation, we try to reduce this penalty and maximize the margin
- As "C" decreases the support vectors increase and vice-versa
- We have to find the right value of "C"

UNIT 5.1.4: SVM Kernels (ML_Support_Vector_Machines.ipynb)

- Kernel trick helps us to project to a higher dimensional space
- Kernel applies the projections to a higher dimensional space
- Kernel choices: "linear", "poly", "rbf" (most powerful), "sigmoid"

UNIT 5.1.5: SVM vs Logistic Regression (ML_Support_Vector_Machines.ipynb)

- SVM resembles logistic regression, they are both linear binary classifiers
- The difference between them is the cost function, this difference occurs in the intermediate zone
- SVM may work better with outliers
- Logistic regression is better in the case where probability is needed
- SVM kernels are slow to train, they don't scale well

UNIT 5.1.6: SVM Regression (ML_Support_Vector_Machines.ipynb)

- We can also use SVM for regression
- For regression we want as many as possible vectors to reside in the margin unlike in classification, we penalize for the vectors outside the margin, the support vectors are now outside the margin
- The greater the distance from the margin the greater the penalty
- We use the "Well Loss" function for SVM regression
- Standard Linear regression uses the "Square Loss" function

UNIT 5.1.7: SVM Lagrangian Dual (ML_Support_Vector_Machines.ipynb)

- I don't understand
- Only the support vectors are going to decide the chosen hyperplane

UNIT 5.1.8: Kernel Trick (ML_Support_Vector_Machines.ipynb)

- I don't understand
- Kernel trick: no need to explicitly expand the function (feature transformation) we the help of "dual formulation"

UNIT 5.1.9: SVM Time Complexity and Multiclass

- It good to understand how the model scales
- Two version of SVM Classifiers: "svm.SVC" (can use kernel tricks) and "svm.LinearSVC" (cannot use kernel tricks). They use different algorithms
- "svm.SVC": $O(n^2*p)$ or (n^3*p)
- "svm.LinearSVC": O(np)
- If you are not bothered about using kernel its best to use "LinearSVC" it scales better than "SVC"
- Play with "class_weight" when we have imbalanced data
- For "one vs all" we train (k)classifiers trains on all the data, for "one vs one" we train (k*k-1/2)classifiers but we only train on a portion(subset) of the data
- SVM uses "one vs one" to handle multi classes because it has better time complexity

UNIT 5.1.10: SVM Tuning Kernels Exercise Part 1 (ML_Support_Vector_Machines.ipynb)

- When using SVM we have to decide if we will use kernel or not, and also the type of kernel to use
- Linear regression is more susceptible to outliers than SVM regressor

UNIT 5.1.11: SVM Tuning Kernels Exercise Part 2 (ML_Support_Vector_Machines.ipynb)

- The randomized took too long, he decided to serialize the model
- "rs.best_score_" and "rs.best_params_"
- SVM don't do so well for large datasets, they don't scale well
- Kernel tricks don't have great time complexities

UNIT 5.1.12: SVM Kernel Approximations (ML_Support_Vector_Machines.ipynb)

- We can use SVM with (SVC/SVR) or without kernel tricks (linear.SVC/SVR)
- With kernels we don't have to explicitly transform our features
- Instead of using kernelized SVM we can use linear SVM with a "kernel approximation"
- Checkout "kernel approximation" documentation in sklearn
- We need to scale our data when using SVM
- Online learning: ability of a model to retrain itself only on new data points
- Models with online learning have a method called "partial_fit"
- "SGDClassifier/Regressor" allows us to do online learning, we can change the "loss" parameter to make it work like a "Linear SVM"

UNIT 5.1.13: SVM Online Learning (ML_Support_Vector_Machines.ipynb)

- In SGD is a lot faster than mormal GD because we use only one data point to get an "approximation" of the gradient
- We don't have memory issues with SGD, and we can use it for online learning
- "alpha" and "C" are inversely related
- Pipelines have no "partial_fit" method (its sad)
- We can create our own partial_fit pipeline

UNIT 5.1.14: SVM Online Learning Pipeline (ML_Support_Vector_Machines.ipynb)

- We should try to build a pipeline class that accommodates our need for "partial_fit" by extending sklearn
- Don't have to reinvent the wheel you can always modify/edit it
- Some transformers like "Standard Scaler" are capable of online learning
- My "partial_fit" was misbehaving

UNIT 5.2.1: Intro to Clustering (ML_Clustering.ipynb)

- Clustering is branch of unsupervised learning, to determine clusters in your dataset
- It does not use "labels", we are not making predictions but grouping into clusters
- We talk about clustering metrics and common algorithms for clustering
- The prediction clustering algorithm do is to assign clusters to the dataset, that were calculated during the "fit" method

UNIT 5.2.2: Metrics for Clustering (ML_Clustering.ipynb)

- We cannot use the metrics we use for “Supervised Machine” learning for our “Clustering” algorithms because we don’t have labels
- Metrics for supervised learning are “objective” but those of clustering are usually more “subjective”
- Two common metrics for clustering: “inertia” and “Silhouette Coefficient”
- We want select clusters with “inertia” as small as possible
- We calculate the “Silhouette Coefficient” for all the data point in the dataset
- If data points are very close to its cluster center and far away from the other clusters center, the “Silhouette Coefficient” is close to 1
- If data points are very far to its cluster center and near to the other clusters center, the “Silhouette Coefficient” is close to -1

UNIT 5.2.3: KMeans Clustering (ML_Clustering.ipynb)

- Kmeans one of the most used “Clustering” algorithm
- The clusters are chosen to reduce the inertia
- Inertia is also called the “sum of square distance function”
- Randomly assign seed centroid > Assign Clusters > Reevaluate the centroid > Assign clusters > Reevaluate the centroid
- It is a greedy algorithm; they are faster to run but the model performance is not the best
- As we increase the number of clusters our “inertia” is going to drop
- We need to scale data inorder to properly use “KMeans” clustering algorithm

UNIT 5.2.4: Elbow Plots (ML_Clustering.ipynb)

- We will plot an “Elbow plot” and look for the point where there is no more significant drop in “inertia”
- Selecting the point from an “Elbow Plot” is subjective
- “kmeans.inertia_”: returns inertia score
- Since it is an unsupervised learning problem, you have to be able to justify your choice

UNIT 5.2.5: Gaussian Mixture Models (ML_Clustering.ipynb)

- Kmeans outputs “isotropic” clusters
- Kmeans does not cluster “elongated shapes” very well
- Gaussian Mixture Models helps to solve this problem
- In GMM cluster definitions are probalistic
- It has two main parameters: “mean”, “variance”
- We are trying to maximize the “likehood function”
- It is slower to train, but more powerful
- People usually use Kmean first, if not satisfied with the result they try out GMM

UNIT 5.2.6: Choosing Cluster Based on Silhouette (ML_Clustering.ipynb)

- Silhouette: measures if the "grasses are cleaner on the other side", it ranges from 1 to -1
- If a data point is not far from its centroid and the closeset centroid the "silhouette score" will be smaller (close to 0)

UNIT 5.2.7: GMM Choosing Number of Components (ML_Clustering.ipynb)

- "Inertia" strives well for "isotropic/ dense clusters" but does not do well for "elongated clusters"
- We will be using "Likehood (log likehood)" as the metric for GMM
- As we increase the clusters the "log likehood" will increase
- You can compare the log likehood or the train and validation sets to determine number of clusters
- "gmm.score(**)": returns the log likehood score
- "gmm.bic(**)": returns the bic score
- GMM does not scale well

SECTION 6 – TIME SERIES AND DIMENSION REDUCTION

UNIT 6.1.1: Intro to Time Series (ML_Time_Series.ipynb)

- Time Series: a sequence of measurements of a variable made over time
- Application of ML to time series is to use the past behavior to make forecast
- Forecasting is a "supervised regression problem"
- Time series made up of 3 components: "Drift", "Seasonality", "Noise"/"residual"
- There are different "drift" models
- Noise: anything left after we remove "drift" and "seasonality"

UNIT 6.1.2: Crossvalidation in Time Series (ML_Time_Series.ipynb)

- In Time series the "order" of the data matters, we have to train on past to forecast the future
- "Sliding Window": model is trained in a fixed window and tested with data in the following window of the same size
- "Forward Chaining": initially similar to sliding window, but the training window increases in size with subsequent fold
- "sklearn.model_selectio.TimeSeriesSplit": to implement forward chain
- Forward chaining takes up more computational time

UNIT 6.1.3: Stationary Signal (ML_Time_Series.ipynb)

- The residual of time series will be “stationary”
- A stationary signal is one which statistical values such as mean do not change with time
- It is easy to predict future values if things like the mean and variance stay the same with time
- “random walk”
- If our residuals are white noise, we cannot forecast the future because what is left is uncorrelated noise (don’t really understand)

UNIT 6.1.4: Modelling Drift (ML_Time_Series.ipynb)

- Order matters with Time Series
- It helps to have domain knowledge of what we are working with

UNIT 6.1.5: Fourier Transforms Part 1 (ML_Time_Series.ipynb)

- We have to have a systematic way to figure out the seasonality of our time series
- Fourier Transform: helps us to determine the predominate frequency
- Most common algorithm used to compute the discrete Fourier Transform is the “from scipy import fftpack”, fast fourier transform
- Fourier transform is taking time series in the “time domain” and giving us frequency in the “frequency domain”, it is symmetric and we have 6 peaks
- It (fourier transform) will have a harder time if we add more “noise”

UNIT 6.1.6: Fourier Transforms Part 2 (ML_Time_Series.ipynb)

- I don’t understand
- The sampling rate helps us measure our signals
- Cause of “aliasing” we only plot the first half of a fourier transform

UNIT 6.1.7: Fourier Components in our model (ML_Time_Series.ipynb)

- We have found a pattern for the time series: once and twice a year dominant frequencies
- So we add the “fourier components” to our model to account for seasonality already containing our “drift”
- Use “FeatureUnion” to combine the drift and seasonality components together in the Pipeline

UNIT 6.1.8: Modelling Noise (ML_Time_Series.ipynb)

- Once we remove “drift” and “seasonality” from our time series model all that is left is “noise/residuals”

- Correlation “1” – linear correlated, an increase in one components corresponds to a linear increase of the other
- Correlation “0” – uncorrelated, higher variable does not necessarily correspond to higher/lower values of the other
- Correlation “-1” – completely linear anti-correlated, increase in one component corresponds to a linear decrease of the other
- “Autocorrelation”: measure of how correlated a signal is with a delayed copy of itself
- “from pandas.plotting import autocorrelation_plot”

UNIT 6.1.9: Moving Statistics (ML_Time_Series.ipynb)

- We can generate features based on past values for each time step
- “Rolling Window Average” and “Exponential Moving Average”
- “Moving Average” help to smooth out our time series residuals
- We can use the autocorrelation plot to determine our window size/half life
- Exponential Moving Average is better considering the computer memory

UNIT 6.1.10: Full Model (ML_Time_Series.ipynb)

- Combine our baseline model (including just seasonality and drift) with our residual model
- The “shift” method
- The residuals are not as correlated as before

UNIT 6.1.11: ARMA and ARIMA

- Statistically based models for time series, they are provided by “statsmodel” python package
- ARMA: Autoregressive and moving average, y(t) dependent on y(t-1)
- MA models are more computationally expensive
- How many past values should we include for our ARMA models, we can use the common “autocorrelation” plot
- “from statsmodels.tsa.ar_model import AR”
- ARMA only work well with stationary process, we can use “difference transformation” to arrive at a stationary process
- ARIMA: Autoregressive Integrated moving average applies the difference transformation in the hopes of generating a stationary process

UNIT 6.1.12: AR Example

- “ar = AR(y)”, “ar = ar.fit(maxlag=**)”, “ar.predict()”

UNIT 6.2.1: Intro to Dimension Reduction (ML_Dimension_Reduction.ipynb)

- Dimension is an unsupervised learning techniques, no labels are used for the process
- The goal of dimension reduction is to reduce the number of features
- We reduce dimension sometime to help reduce size, visualize dataset, faster training and predicting for supervised learning, better truncated set of new features to represent our data
- "regularization to reduce overfitting"
- It helps to generate uncorrelated features
- We will discuss 3 commonly used techniques for dimension reduction

UNIT 6.2.2: Math of Projections (ML_Dimension_Reduction.ipynb)

- Projecting our data from initial space to a new space
- We trying to reduce the dimensions so the rows remain the same but the features (columns) are reduced
- The objective function is typically constructing a truncated form of the data that retains the majority of the information in our data

UNIT 6.2.3: PCA (ML_Dimension_Reduction.ipynb)

- Principal Component Analysis (PCA) is a dimension reduction technique, it takes a possibly correlated features and generates a new set of features that are uncorrelated
- PCA is rarely used when the dimension of the data set is already low
- "np.corrcoef()"

UNIT 6.2.4: PCA in Scikit Learn (ML_Dimension_Reduction.ipynb)

- Dimension Reduction algorithms are "Transformers"
- PCA centers the data but it "does not scale it", it is advised to scale the features to allow PCA to work properly
- "Sketch out a Workflow"
- The features in PCA are selected based on their importance in order to keep most of the information of our data
- "fit(**)", solves for the projection matrix
- "pca.components_": returns a transpose of the Projection matrix
- "pca.explained_variance_/ratio_": variance captured by PCs/returns the ratio of variance captured
- "pca.inverse_transform(Truncated data)": it returns the original space but it has lost some variations of the original data

UNIT 6.2.5: PCA Implementation Details (ML_Dimension_Reduction.ipynb)

- We try minimize the difference between original space and the new projected space
- “Single Value Approximation” (SVD) is one of the several algorithms we use to solve for principal components

UNIT 6.2.6: Choosing the number of Components (ML_Dimension_Reduction.ipynb)

- One of the best way is to construct a plot of the cumulative explained variance versus the number of component (similar to an elbow plot)
- Choosing the number of components is still overall a subjective approach
- When we pass a float in our “PCA(n_components = float)”, we asking for the number of features that will achieve that explained variance

UNIT 6.2.7: Truncated SVD (ML_Dimension_Reduction.ipynb)

- In order to apply PCA the data needs to be centered i.e. the features need to have zero means
- We cannot center sparse matrix because after centering it, it will not return a sparse matrix which might cause memory issues
- Sparse matrix is common in the realm of NLP
- As an alternative to PCA we can use the “TruncatedSVD” class, it does need to center the data before finding the principal components
- “from sklearn.decomposition import TruncatedSVD”

UNIT 6.2.8: NMF (ML_Dimension_Reduction.ipynb)

- In NLP, bag of words model yields a matrix of only non-negative values
- Non-negative Matrix Factorization (NMF): a variation of PCA that has the added constraint that derived matrices are non-negative
- In dimension reduction, our new features are just a composition of old features
- The new features gotten when applying dimension reduction to bag of words output can be called “Topics”

UNIT 6.2.9: Using PCA with Supervised ML (ML_Dimension_Reduction.ipynb)

- If we reduce the number of features, we will have a faster training time
- Use the elbow plot as a guide to choose optimal number of PCs
- Sometimes the cost of faster training is accuracy

UNIT 6.2.10: PCA for Visualization (ML_Dimension_Reduction.ipynb)

- We use dimension reduction to visualize high dimension data sets

UNIT 6.2.11: NMF Exercise Part 1 (ML_Dimension_Reduction.ipynb)

- Use of NMF is also useful in the field of Image Classification
- For image classification, each pixel location counts as a feature (when working with grayscale images), it gets more complicated when working with colored images

UNIT 6.2.12: NMF Exercise Part 2 (ML_Dimension_Reduction.ipynb)

- Elbow plot to get optimal number of PCs
- Sometimes the cost of faster training is accuracy
- Using "memory cache" to reduce time complexity

UNIT 6.2.13: Variants of PCA (ML_Dimension_Reduction.ipynb)

- Somethings using dimension reduction will lead to a better generalized model usually where our model is prone to overfitting
- Time complexity with PCA: O(n^2 *p) + O(n^3)
- Randomized PCA: O(m^2 *p) + O(m^3), m = number of components we are outputting
- PCA is not capable of online learning
- "ImcrementalPCA": a variant of PCA that is capable of online learning
- "KernalPCA": we use "kernel trick" to solve for PCs, more computational expensive

SECTION 7 – ANOMALY DETECTION

UNIT 7.1.1: Intro to Anomaly Detection (ML_Anomaly_Detection.ipynb)

- Anomaly Detection: process of finding abnormal and unusual values in our data set
- "Outlier Detection": identify observations that deviate substantially from the rest, it is trained with datasets that include outliers
- "Novelty Detection": identify novel points by training model with a data set that is not "polluted" with outliers, it learns a boundary and encompasses the normal/regular points
- Outlier and Novelty Detection are usually lumped together
- Anomaly Detection are "unsupervised learners"
- "One class SVM": novelty detection
- "isolation forest": outlier detection
- "fit()", "predict()", "decision_function()": key methods for anomaly detection algorithms

UNIT 7.1.2: One Class SVM (ML_Anomaly_Detection.ipynb)

- "One class SVM" is a tweaked SVM classifier used to serve novelty detection applications

- The algorithm task is to locate a hyperplane in the space that best separates the data from the origin
- As we reduce the hyper parameter “v” our “false positive” reduces

UNIT 7.1.3: Isolation Forest (ML_Anomaly_Detection.ipynb)

- It is an “outlier detection” that uses decision trees
- Decision tree groups our observations into bins (box them in)
- Group every single observation into a node, it grows until all points have been isolated
- The outliers are easy to isolate (they isolate fast) because they are far from everything else
- “how many node splits did it take to isolate the point?”
- The key hyper parameters: “n_estimators”, “contamination” (fraction of outliers in the data set)
- It is computationally cheap
- “score”: a negative score means an outlier and vice versa
- With increase in “contamination” the decision boundary shrinks

UNIT 7.1.4: Comparison Between One-Class SVM and Isolation Forest (ML_Anomaly_Detection.ipynb)

- Both models are capable of modelling multi-modal datasets (data set with more than one cluster areas)
- Isolation Forest is faster to train (since the splits are random)
- They both inherit the pros and cons of their parent algorithm
- There are other outlier and novelty detection algorithms in sklearn

UNIT 7.1.5: Intro to Case Study (ML_Anomaly_Detection.ipynb)

- Anomaly detection can be applied to time series
- Observations that are flagged as anomalous can be analyzed
- Check for drift > Remove drift > Check for seasonality > Remove seasonality > Work on residual

UNIT 7.1.6: Initial Baseline Model Part 1 (ML_Anomaly_Detection.ipynb)

- Index Selector > Convert index to time > Construct Fourier Components
- Unix Time
- Sklearn is built on numpy
- I don’t really understand

UNIT 7.1.7: Initial Baseline Model Part 2 (ML_Anomaly_Detection.ipynb)

- Index Selector > (Convert index to time > Fourier) + (day of week > OHE) > Predictor > y
- Workflow diagrams are very important

UNIT 7.1.8: Full Baseline Model (ML_Anomaly_Detection.ipynb)

- To improve our model, we need to add noise based models (residual model)

UNIT 7.1.9: Z-Score Detection (ML_Anomaly_Detection.ipynb)

- How big of a deviation is big? We can use Z-Score for this
- Z-Score is a relative measure of how far away a value is from the mean normalized by the standard deviation
- There is no right point to uses as out Z-score cutoff, it is to some extent subjective

UNIT 7.1.10: Rolling Z-Score Detection (ML_Anomaly_Detection.ipynb)

- Rolling Z-Score calculates the z-score on a window of observations rather than the entire time history
- It is more adaptive to recent changes in the process, it reflects the fact that it is better to use recent values to judge an observation
- We can also use Exponential Weight Decay

UNIT 7.1.11: Using External Features Initial Model (ML_Anomaly_Detection.ipynb)

- Feature Union can combine two parallel pipelines together

UNIT 7.1.12: Using External Features Tuning the Model (ML_Anomaly_Detection.ipynb)

UNIT 7.1.13: Packaging the Time Series Anomaly Detector ((ML_Anomaly_Detection.ipynb)

- Sklearn version 0.20 upwards have an “OutlierMixin” class

SECTION 8 – MODEL DEPLOYMENT

UNIT 8.1.1: Model Considerations (Sentiment Analysis)

- Let us create a web app for Machine Learning model
- Create a sentiment analysis model
- Supervised/Unsupervised > (if supervised) > Regression/Classification

- The hardest part of our task is usually “finding data”, the data we have can influence our model considerations
- “thinknook.com”: site to get twitter sentiment analysis dataset
- Never too early to start drawing our Workflow diagram
- “Probabilistic Models”: capable of predicting probabilities
- Usually use “naives bayes model” as a baseline model

UNIT 8.1.2: Model Development (Sentiment Analysis)

- NLTK package in python
- Naïve Baye is capable of online learning, therefore we can train it batch wise
- Need a way to persist our model (use it predict without retraining it), can be done by “pickling” it or “dlling” it (serializing it)

UNIT 8.1.3: Flask App Local Development

- Flask is a micro framework for Python
- “GET request”: in this any data we are sending to the server is located in the url

UNIT 8.1.4: GET Requests

UNIT 8.1.5: Making GET Requests with Model

- Always make sure that the version of packages you used to build and serialize your model are the same with the ones you use in your virtual environment for the Flask app

UNIT 8.1.6: Using our Model with Twitter Web API

- You need authentication to use Twitter API
- Check out “developer twitter” website

UNIT 8.1.7: POST Requests and Flask Templates

- In POST requests the data we pass is in “request” itself unlike GET requests where the data is in the URL

UNIT 8.1.8: Preparing for Deployment to the Web

- We will use a cloud platform service to deploy our app to a webpage
- Create a “runtime.txt”: specify version of python we are using
- Create a “requirement.txt”: specify version of packages used
- Create a “Procfile” file
- Make the folder a git repository (git init) cause of memory issues remove the not needed files

UNIT 8.1.9: Deploying our App to the Web with Heroku

- Install heroku > launch Command Prompt
- "Signing to Heroku": heroku login (command line) > heroku create
- "git add ." > "git commit –m "***" " > "git push heroku master > "heroku open"
- "heroku logs": helps to debug error messages
- https://young-shelf-21980.herokuapp.com/index

UNIT 8.2.1: Rethinking Model Tuning

- GridSearchCV is computationally expensive, tries all possible combination, there is no smart search, its more of a trial and error method
- RandomSearchCV is like "random sampling", there is no smart search, its more of a trial and error method
- "Bayesian Search"

UNIT 8.2.2: Intro to Bayes Theorem

- Bayes Theorem helps to update a prior belief based on some observations
- P(A given B) = P(B given A)*P(A) / P(B)
- Number of heads and unfair coin analogy

UNIT 8.2.3: Bayesian Inference

- As it takes in data it updates it believe system
- We can use something like Bayes Theorem to tune our model

UNIT 8.2.4: Bayesian Optimization (Bayes-theorem.ipynb)

- Need to install the "scikit-optimize" package, we will be using its "BayesSearchCV" class
- Not much restrictions with BayesSearchCV, no need for step sizes
- It has to balance uncertainty with exploration
- It will keep the best one of all the iterations

UNIT 8.2.5: End of Course Material

- How to download the course content: download as "zip file" or use command line
- Setup Jupyter Notebook on your local machine, you can download "Anaconda"
- Visit "Hackerrank"
- Checkout Sklearn Documentations

- Check out the ebooks “An introduction to statistical learning”, “The Elements of Statistical Learning”
- Check out “Machine Learning Yearning” By Andrew Ng