Key takeaways from the Datacamp xgboost course

Intro:

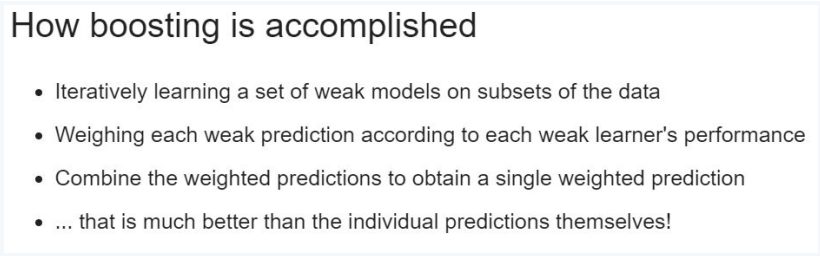
AUC-ROC Definition: It is simply the probability of a randomly chosen positive data point has a higher ranking than a randomly chosen negative data point.

Why XGBoost?
Speed
Written in C++
Can be used with several APIs: Python, Julia, R….
Parallelisable
State of the Art ML Technique
Beats all the algos.

Interesting thread on reddit on why XGBoost is still not as popular as deep learning, despite winning kaggle competitions
https://www.reddit.com/r/MachineLearning/comments/53o28t/why_isnt_xgboost_a_more_popular_research_topic/

For classification, are xgboost and logreg equivalent, for a certain loss function?
https://datascience.stackexchange.com/questions/18081/gradient-boosting-vs-logistic-regression-for-boolean-features
Yes, when features are all binary and independent!

Not an algorithm in itself. It is a concept can be applied to many ML algorithm
It is “Meta-algorithm”
Many weak learners into a strong learner.
Weak: A decision tree that can predict slightly better than a chance (50%)



Some basics of Gradient Boosting from another ppt:
Gradient Boosting: “Gradient” used instead of “residuals” even though first came out by trying to fit additional trees to residuals, because gradient finding is more general and can be applied to other loss functions. Why square loss is not good enough? Because outliers are heavily punished, it tries to fit model appropriate for outliers as well.
Taking gradients instead of residuals (by defining a loss function other than square), effect of outliers on the model is less.

Loss Functions for Regression Problem

- Absolute loss (more robust to outliers)

$$L(y, F) = |y - F|$$

- Huber loss (more robust to outliers)

$$L(y, F) = \begin{cases} \frac{1}{2}(y - F)^2 & |y - F| \le \delta \\ \delta(|y - F| - \delta/2) & |y - F| > \delta \end{cases}$$

y_i	0.5	1.2	2	5*
$F(x_i)$	0.6	1.4	1.5	1.7
Square loss	0.005	0.02	0.125	5.445
Absolute loss	0.1	0.2	0.5	3.3
Huber loss($\delta = 0.5$)	0.005	0.02	0.125	1.525

General procedure for Gradient Boosting F(x) = y

Regression with loss function L: general procedure

Give any differentiable loss function L

start with an initial model, say $F(x) = \frac{\sum_{i=1}^{n} y_i}{n}$
iterate until converge:
  calculate negative gradients $-g(x_i) = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$
  fit a regression tree h to negative gradients $-g(x_i)$
  $F := F + \rho h$

Summary for Regression

Summary of the Section

- Fit an additive model $F = \sum_t \rho_t h_t$ in a forward stage-wise manner.
- In each stage, introduce a new regression tree h to compensate the shortcomings of existing model.
- The "shortcomings" are identified by negative gradients.
- For any loss function, we can derive a gradient boosting algorithm.
- Absolute loss and Huber loss are more robust to outliers than square loss.

Datacamp continuation:
Cross validation builtin in xgboost xgb.Dmatrix
xgb.cv function
Params is a dictionary

Cross-validation in XGBoost example

```
In [1]: import xgboost as xgb

In [2]: import pandas as pd

In [3]: class_data = pd.read_csv("classification_data.csv")

In [4]: churn_dmatrix = xgb.DMatrix(data=churn_data.iloc[:,:-1],
        label=churn_data.month_5_still_here)

In [5]: params={"objective":"binary:logistic","max_depth":4}

In [6]: cv_results = xgb.cv(dtrain=churn_dmatrix, params=params, nfold=4,
        num_boost_round=10, metrics="error", as_pandas=True)

In [7]: print("Accuracy: %f" %((1-cv_results["test-error-mean"]).iloc[-1]))
Accuracy: 0.88315
```

When to use XGBoost

- You have a large number of training samples
  - Greater than 1000 training samples and less 100 features
  - The number of features < number of training samples
- You have a mixture of categorical and numeric features
  - Or just numeric features

When to NOT use XGBoost

- Image recognition
- Computer vision
- Natural language processing and understanding problems
- When the number of training samples is significantly smaller than the number of features

Deep learning More Suitable for first 3 bullet points

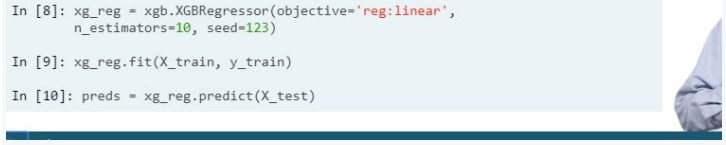
Common Loss functions: (objective argument in xgboost)
Regression: “reg:linear”
Classification: “reg:logistic”
Ranking/Probability: “binary:logistic”

Each weak learner gives slightly better prediction than chance
Uniformly bad predictions from weak learners cancel out, and good predictions get amplified.

Trees and Linear base learners possible.
Tree example

Trees as Base Learners example: Scikit-learn API

```
In [1]: import xgboost as xgb

In [2]: import pandas as pd

In [3]: import numpy as np

In [4]: from sklearn.model_selection import train_test_split

In [5]: boston_data = pd.read_csv("boston_housing.csv")

In [6]: X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

In [7]: X_train, X_test, y_train, y_test= train_test_split(X, y,
        test_size=0.2, random_state=123)

In [8]: xg_reg = xgb.XGBRegressor(objective='reg:linear',
        n_estimators=10, seed=123)

In [9]: xg_reg.fit(X_train, y_train)

In [10]: preds = xg_reg.predict(X_test)
```



Linear Base Learners Example: Learning API Only

```
In [1]: import xgboost as xgb

In [2]: import pandas as pd

In [3]: import numpy as np

In [4]: from sklearn.model_selection import train_test_split

In [5]: boston_data = pd.read_csv("boston_housing.csv")

In [6]: X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

In [7]: X_train, X_test, y_train, y_test= train_test_split(X, y,
        test_size=0.2, random_state=123)

In [8]: DM_train = xgb.DMatrix(data=X_train,label=y_train)

In [9]: DM_test =  xgb.DMatrix(data=X_test,label=y_test)

In [10]: params = {"booster":"gblinear","objective":"reg:linear"}

In [11]: xg_reg = xgb.train(params = params, dtrain=DM_train,
         num_boost_round=10)
```

Tree based base learners
Penalizing models more as they become more complex is called Regularization

Gamma: Higher values lead to fewer splits. Gamma is the minimum loss reduction allowed for a split to occur.
Alpha: l1 regularization (el1), higher means more regularization. Controls leaf weights. Many leaf weight can go to zero
Lambda: l2 regularization. Smoother regularization

Base Learners in XGBoost

- Linear Base Learner:
  - Sum of linear terms
  - Boosted model is weighted sum of linear models (thus is itself linear)
  - Rarely used
- Tree Base Learner:
  - Decision tree
  - Boosted model is weighted sum of decision trees (nonlinear)
  - Almost exclusively used in XGBoost

Linear Rarely used because it is similar to regularized linear or logistic regression model.

Creating DataFrames from multiple equal-length lists

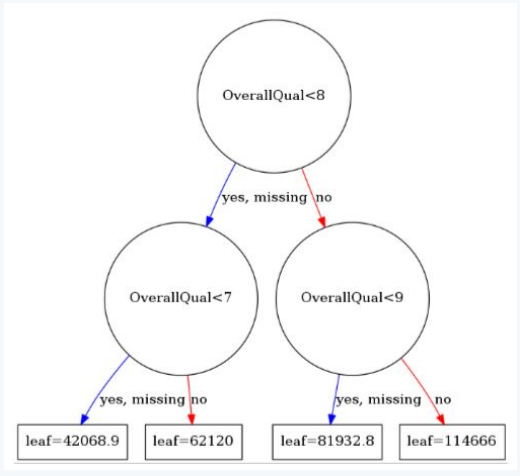
- `pd.DataFrame(list(zip(list1,list2)),columns=["list1","list2"]))`
- `zip` creates a `generator` of parallel values:
  - `zip([1,2,3],["a","b""c"])` = `[1,"a"],[2,"b"],[3,"c"]`
  - `generators` need to be completely instantiated before they can be used in `DataFrame` objects
- `list()` instantiates the full generator and passing that into the `DataFrame` converts the whole expression

Regularized xgboost code

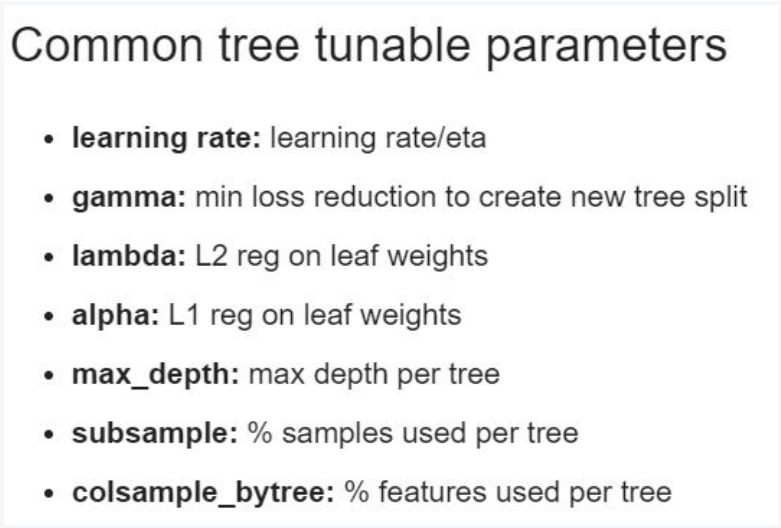
```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

reg_params = [1, 10, 100]

# Create the initial parameter dictionary for varying l2 strength: params
params = {"objective":"reg:linear","max_depth":3}
# Create an empty list for storing rmses as a function of l2 complexity
rmses_l2 = []

# Iterate over reg_params
for reg in reg_params:

    # Update l2 strength
    params["lambda"] = reg

    # Pass this updated param dictionary into cv
    cv_results_rmse = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2, num_boost_round=5, metrics="rmse",
as_pandas=True, seed=123)

    # Append best rmse (final round) to rmses_l2
    rmses_l2.append(cv_results_rmse["test-rmse-mean"].tail(1).values[0])
# Look at best rmse per l2 param
print("Best rmse as a function of l2:")
print(pd.DataFrame(list(zip(reg_params, rmses_l2)), columns=["l2", "rmse"]))
```

Plotting the tree:

```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":2}

# Train the model: xg_reg
xg_reg = xgb.train(params=params, dtrain=housing_dmatrix, num_boost_round=10)

# Plot the first tree
xgb.plot_tree(xg_reg,num_trees=0)
plt.show()
```



With rankdir=LR for plot_tree, it plots horizontal



For Linear base learners, it is lambda, alpha and lambda_bias-l2 regularization term on bias. Only 3 parameters. n_estimators is common to both.

```
In [1]: import pandas as pd
In [2]: import xgboost as xgb
In [3]: import numpy as np
In [4]: from sklearn.model_selection import GridSearchCV

In [5]: housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
In [6]: X, y = housing_data[housing_data.columns.tolist()[:-1]],
   ...: housing_data[housing_data.columns.tolist()[-1]
In [7]: housing_dmatrix = xgb.DMatrix(data=X,label=y)

In [8]: gbm_param_grid = {
   ...: 'learning_rate': [0.01,0.1,0.5,0.9],
   ...: 'n_estimators': [200],
   ...: 'subsample': [0.3, 0.5, 0.9]}

In [9]: gbm = xgb.XGBRegressor()
In [10]: grid_mse = GridSearchCV(estimator=gbm,
    ...: param_grid=gbm_param_grid,
    ...: scoring='neg_mean_squared_error', cv=4, verbose=1)
In [11]: grid_mse.fit(X, y)

In [12]: print("Best parameters found: ",grid_mse.best_params_)
Best parameters found: {'learning_rate': 0.1,
```

Randomized search: 400 combos are there, but n_iter limits them to 25

```
In [1]: import pandas as pd
In [2]: import xgboost as xgb
In [3]: import numpy as np
In [4]: from sklearn.model_selection import RandomizedSearchCV
In [5]: housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
In [6]: X,y = housing_data[housing_data.columns.tolist()[:-1]],
   ...: housing_data[housing_data.columns.tolist()[-1]]
In [7]: housing_dmatrix = xgb.DMatrix(data=X,label=y)

In [8]: gbm_param_grid = {
   ...: 'learning_rate': np.arange(0.05,1.05,.05),
   ...: 'n_estimators': [200],
   ...: 'subsample': np.arange(0.05,1.05,.05)}

In [9]: gbm = xgb.XGBRegressor()
In [10]: randomized_mse = RandomizedSearchCV(estimator=gbm,
    ...: param_distributions=gbm_param_grid, n_iter=25,
    ...: scoring='neg_mean_squared_error', cv=4, verbose=1)
In [11]: randomized_mse.fit(X, y)
```

Limitations

Grid Search and Random Search Limitations

- Grid Search
  - Number of models you must build with every additional new parameter grows very quickly
- Random Search
  - Parameter space to explore can be massive
  - Randomly jumping throughout the space looking for a "best" result becomes a waiting game

Pipeline

```
In [1]: import pandas as pd
   ...: from sklearn.ensemble import RandomForestRegressor
   ...: import numpy as np
   ...: from sklearn.preprocessing import StandardScaler
   ...: from sklearn.pipeline import Pipeline
   ...: from sklearn.model_selection import cross_val_score

In [2]: names = ["crime","zone","industry","charles",
   ...: "no","rooms","age", "distance",
   ...: "radial","tax","pupil","aam","lower","med_price"]

In [3]: data = pd.read_csv("boston_housing.csv",names=names)

In [4]: X, y = data.iloc[:,:-1], data.iloc[:,-1]

In [5]: rf_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()),
   ...: ("rf_model",RandomForestRegressor())]

In [6]: scores = cross_val_score(rf_pipeline,X,y,
   ...: scoring="neg_mean_squared_error",cv=10)
```

Negative mean errors dont exist, they are calculated as -mean squared error

Building Pipelines

For one hot encoding, DictVectorizer of feature extraction library can be used:

Scikit-Learn Pipeline Example With XGBoost

```
In [1]: import pandas as pd
   ...: import xgboost as xgb
   ...: import numpy as np
   ...: from sklearn.preprocessing import StandardScaler
   ...: from sklearn.pipeline import Pipeline
   ...: from sklearn.model_selection import cross_val_score

In [2]: names = ["crime","zone","industry","charles","no",
   ...: "rooms","age", "distance","radial","tax",
   ...: "pupil","aam","lower","med_price"]
In [3]: data = pd.read_csv("boston_housing.csv",names=names)
In [4]: X, y = data.iloc[:,:-1], data.iloc[:,-1]

In [5]: xgb_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()),
   ...: ("xgb_model",xgb.XGBRegressor())]
In [6]: scores = cross_val_score(xgb_pipeline, X, y,
   ...:scoring="neg_mean_squared_error",cv=10)

In [7]: final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
In [8]: print("Final XGB RMSE:", final_avg_rmse)
Final RMSE: 4.02719593323
```

There is a library called sklearn_pandas

- DataFrameMapper: Interoperability between dataframe and sklearn object
- CategoricalImputer: Allows imputation categorical variables without converting to integers

Sklearn.preprocessing has Imputer (For numeric feature imputation)
Sklearn.pipeline Has FeatureUnion: To combine preprocessed Features.

```
In [1]: import pandas as pd
   ...: import xgboost as xgb
   ...: import numpy as np
   ...: from sklearn.preprocessing import StandardScaler
   ...: from sklearn.pipeline import Pipeline
   ...: from sklearn.model_selection import RandomizedSearchCV

In [2]: names = ["crime","zone","industry","charles","no",
   ...: "rooms","age", "distance","radial","tax",
   ...: "pupil","aam","lower","med_price"]
In [3]: data = pd.read_csv("boston_housing.csv",names=names)
In [4]: X, y = data.iloc[:,:-1],data.iloc[:,-1]
In [5]: xgb_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()), ("xgb_model",xgb.XGBRegressor())]

In [6]: gbm_param_grid = {
   ...:     'xgb_model__subsample': np.arange(.05, 1, .05),
   ...:     'xgb_model__max_depth': np.arange(3,20,1),
   ...:     'xgb_model__colsample_bytree': np.arange(.1,1.05,.05) }

In [7]: randomized_neg_mse = RandomizedSearchCV(estimator=xgb_pipeline,
   ...: param_distributions=gbm_param_grid, n_iter=10,
   ...: scoring='neg_mean_squared_error', cv=4)
```

Ranking Recommendation
Sophisticated Hyperparameter tuning strategies: Bayesian optimization
Ensembling other methods with XGBoost (although XGBoost is ensemble)