IBM MACHINE LEARNING

Supervised Regression

$y_p = f(\Omega, x)$

observations – row
features – column

$x \equiv$ input
$y_p \equiv$ predicts
$\Omega \equiv$ parameter of model or weights
hyperparameter – defines models, set during training
$J(y, y_p) \equiv$ loss
Update: use x and y, choose Ω to minize loss J

Interpretate vs Predict approach.

Linear Regression:

$y_\beta(x) = \beta_0 + \beta_1 x$

residuals, $\min \sum \left((\beta_0 + \beta_1 x_{obs}^{(i)}) - y_{obs}^{(i)} \right)^2$ MSE → L2 norm

minimize error, $J(\beta_0, \beta_1) = \frac{1}{2m} \sum \left((y_{pred}^{(i)}) - (y_{obs}^{(i)}) \right)^2$

$SSE = \sum \left(y_\beta(x^{(i)}) - y_{obs}^{(i)} \right)^2$

$TSS = \sum \left(\bar{y}_{obs} - y_{obs}^{(i)} \right)^2$

$R^2 = 1 - \frac{SSE}{TSS}$

$boxcox(y_i) = \frac{y_i^\lambda - 1}{\lambda}$ scipy.stats.boxcox
scipy.stats.mstats.normaltest

```
pf = PolynomialFeatures(degree=4, include_bias=False)
X_pf = pf.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                   test_size=0.3, random_state)

ss = StandardScaler()
X_train_ss = ss.fit_transform(X_train)
X_test_ss = ss.transform(X_test)

lr = LinearRegression()
lr.fit(X_train_ss, y_train)
y_pred = lr.predict(X_test_ss)
r2_score(y_test, y_pred)
```

Training $\xrightarrow[y\_train]{X\_train}$ model(X_train, y_train).fit() → model

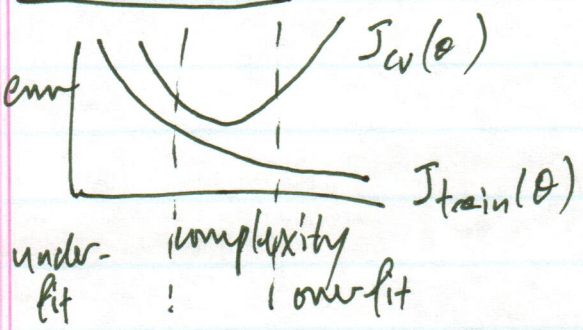
Test $\xrightarrow{X\_test}$ model.predict(X_test) → y_predict

↳ → error_metric(y_test, y_pred) → error/metric

```
train_test_split()
ShuffleSplit()
StratifiedShuffleSplit()

OneHotEncoder()
pd.get_Dummies()
```

```
ax = sns.scatterplot(y_test, y_pred)
ax.set(xlabel='', ylabel='', title='')
```

Cross validation:



k-fold cross validation
Leave one out CV
Stratified CV

```
cross_val = cross_val_clone(model, X_data, y_data, cv=k,
                            scoring='neg_mean_squared_error')
```

```
kf = KFold(shuffle=True, random_state, n_splits=n)
for train_index, test_index in kf.split(X):
    X_train, X_test, y_train, y_test = (X.iloc[train_index, :],
                                        X.iloc[test_index, :],
                                        y[train_index],
                                        y[test_index])

    lr.fit(X_train, y_train)
    y_pred = lr.predict(X_test)
    score = r2_score(y_test.values, y_pred)
    scores.append(score)
```

```
estimator = Pipeline([('scaler', StandardScaler()),
                      ('regression', LinearRegression())])
estimator.fit(X_train, y_train),
estimator.predict(X_test)
```

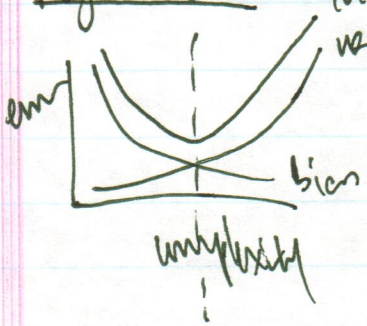
```
prediction = cross_val_predict(estimator, X, y, cv=kf)
r2_score(y, prediction)
~~np.mean(prediction)~~
```

```
Lasso(alpha=α)
 smaller α, more complex, less regularizati
                                    .get_feature_names(input_features=X.columns)
estimator.named_step["regression"].coef_
```

```
Ridge(alpha=α)
 smaller α, more complex, less regularizati
```

```
params = {'pf__degree': [1, 2, 3],
          'ridge__α': [np.geomspace[4, 20, 30]]
         }
grid = GridSearchCV(estimator, params, cv=kf)
grid.fit(X, y), grid.predict(X)                                    []
grid.best_score_, grid.best_params_, grid.best_estimator_.named_steps.
grid.cv_results_                                                   .coef_
```

Regularization: total error, variance, error, bias, complexity



$M(w) + \lambda R(w)$ $M(w)$: model error

λ : regularization parameter / penalty

$R(w)$: regularization function

more regularization, larger α, less complex model, more bias

less regularization, smaller α, more complex model, more variance

if overfit, too high variance, regularization reduce error, variance.

Ridge (L2):

$$J(\beta_0, \beta_j) = \sum_i \left(y_i - \beta_0 - \sum_j \beta_j x_{ij}\right)^2 + \lambda \sum_j \beta_j^2 = \underset{\text{squares}}{\text{residual sum of}} + \lambda \sum_j \beta_j^2$$

- λ applied to β_j^2
- shrinking effect of β towards 0
- improves bias, reduces variance

$$J(\beta_0, \beta_i) = \frac{1}{2m} \sum_i \left(\left(\beta_0 + \beta_1 x_{obs}^{(i)}\right) - y_{obs}^{(i)}\right)^2 + \lambda \sum_j^K \beta_j^2$$

larger weights penalize more due to squaring.

Lasso (L1):

$$\sum_i \left(y_i - \beta_0 - \sum_j \beta_j x_{ij}\right)^2 + \lambda \sum_j \beta_j = RSS + \lambda \sum_j \beta_j$$

- λ proportional to β_j
- increasing λ increases bias, lowers variance
- perform feature selection on coefficients, set to zero

combo

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_i \left(\left(\beta_0 + \beta_i x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2 + \gamma \sum_j |\beta_j|$$

slower to converge than Ridge

Elastic Net:

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_i \left(\left(\beta_0 + \beta_i x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2 + \lambda \sum_j \left(\alpha \beta_j^2 + (1-\alpha) |\beta_j| \right)$$

L2 Ridge L1 Lasso

RFE:

repeatedly measures feature importance, removes less important features.

alphas = [ ]
ridgeCV = RidgeCV(alphas = alphas, cv = 4).fit(X_train, y_train)
ridgeCV.alpha_, ridgeCV_rmse = rmse(y_test, ridgeCV.predict(X_test))

lassoCV = LassoCV(alphas, max_iter, cv).fit( )
~~lasso~~ lassoCV.alpha_
lassoCV.coef_

elasticNetCV = ElasticNetCV(alphas, l1_ratio = [ ], max_iter).fit(,)
elasticNetCV.alpha_, .l1_ratio_,

Supervised ~~Regression~~ classification

- Logistic Regression:

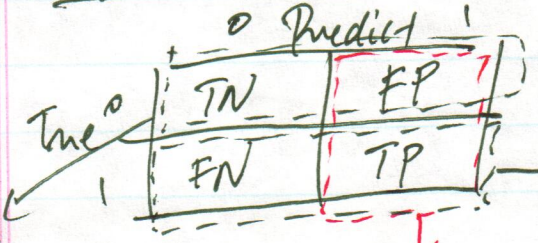
$$P(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x+\varepsilon)}} = \frac{e^{(\beta_0+\beta_1 x)}}{1+e^{(\beta_0+\beta_1 x)}}$$

odds ratio, $\log\left(\frac{P(x)}{1-P(x)}\right) = \beta_0 + \beta_1 x$

LR = LogisticRegression(penalty='l2', C=10.0) (Ridge 大)
LR.fit(,)
LR.predict( )
LR.coef_

LogisticRegressionCV(penalty='l2' 'l1', Cs=[ ], cv=4).fit(,)

Confusion Matrix:



$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FN+FP}$$

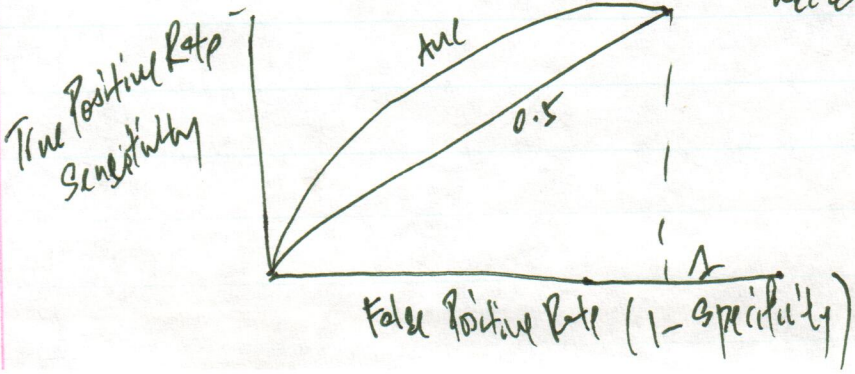
$$\text{Recall} = \frac{TP}{TP+FN}$$

$$\text{Precision} = \frac{TP}{TP+FP}$$

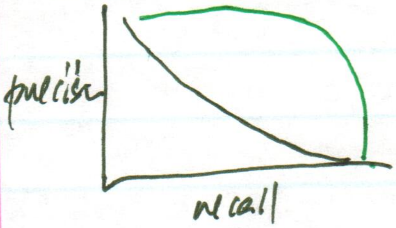
specificity ~~sensitivity~~ $= \frac{TN}{TN+FP}$

$$F1 = 2\,\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Reciever Operating Characteristic (ROC)



Precision - Recall curve - imbalanced dataset



accuracy_score(y_test, y_pred)

precision_score(,)

recall_score(,)

f1_score(,)

confusion_matrix

roc_curve, roc_auc_score

precision_recall_curve

lr_l1 = LogisticRegressionCV(Cs=[ ] cv=4, penalty='l1', 'l2'
solver='liblinear'). fit(,)

lr_l1.predict( )

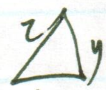
lr_l1.predict_proba( )

auc = roc_auc_score(label_binarize(y_test, classes=[0,1,2,3,4,5]),
y_prob, average='weighted')

cm = confusion_matrix(y_test, y_pred)

sns.heatmap(cm)

<u>KNN:</u>

manhattan distance 

x+y L1   Euclidean distance, z

KNN = KNeighborsClassifier (Regression) (n_neighbors=3, similarity, weights)

KNN.fit(,)

KNN.predict( )

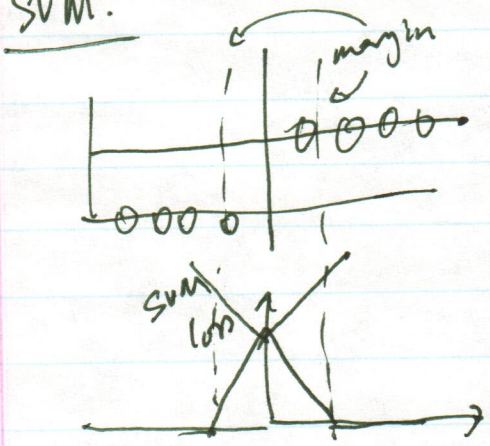
high k, high bias, low k, high variance

categorical } LabelEncoder or pd.get_dummies(df, columns, drop_first=True)
ordinal } OrdinalEncoder
binary } LabelBinarizer

StandScaler or MinMaxScaler
scaler.fit_transform(df[column])

for k in range(1, max_k):
KNN.fit(,)
y_pred = knn.predict( )
classification_report(y_test, y_pred)
accuracy_score(,)
f1_score(,)
cm(,)

SVM:



$J(\beta_i) = SVMCost(\beta_i) + \frac{1}{C}\sum_i \beta_i$

Smaller C, more regularization, more simple model

SVM
linearSVC = LinearSVC(penalty='l2', C=10)
linearSVC.fit(,)
linearSVC.predict( )

Lower γ, C, more regularization, simpler model

rbfSVC = SVC(kernel='rbf', gamma=1.0, C=10)
rbfSVC.fit(,), rbfSVC.predict( )

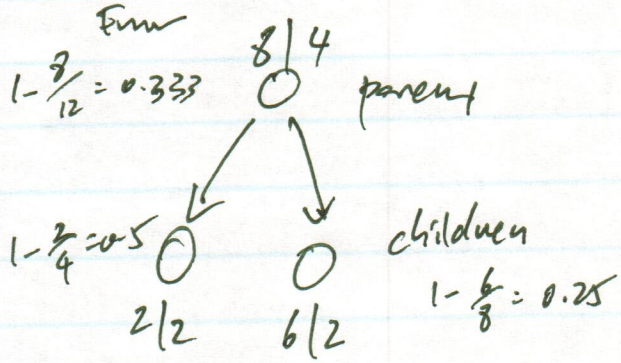
Features	Data	Model choice
many (>10k)	small (<1k rows)	simple, logistic, linear SVC
few (<100)	medium (10k rows)	SVC w RBF
few (<100)	large (>100k rows)	add features, logistic, linear SVC, kernel approx.

```
nystrom_svc = Nystroem(kernel='rbf', gamma, n_components)
X_train = nystrom_svc.fit_transform(X_train)
X_test = nystrom_svc.transform(X_test)

rbf_sampler = RBFSampler(gamma, n_components)

sns.pairplot(    )
```

similar

Decision Tree:

keep splitting until
1. only pure class remains
2. max_depth reached
3. performance metric reached

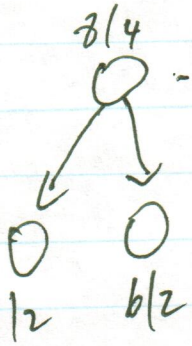


classification error during split
$E(t) = 1 - \max[p(i|t)]$

Classification error change
$= 0.333 - \frac{4}{12} \times 0.5 - \frac{8}{12} \times 0.25$
$= 0$
no further split

Split based on Entropy:

$$H(t) = -\sum_i p(i|t) \log_2 [p(i|t)]$$



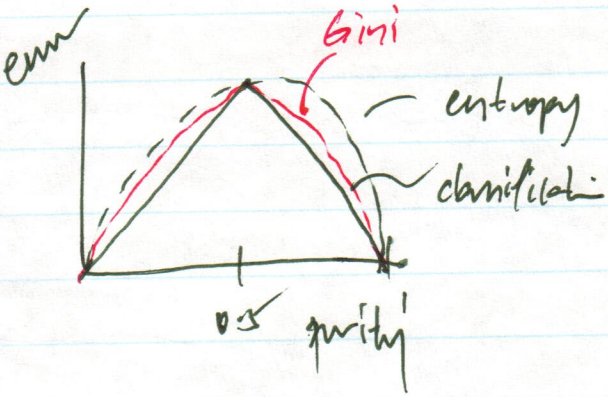
$-\frac{8}{12}\log_2\left(\frac{8}{12}\right) - \frac{4}{12}\log_2\left(\frac{4}{12}\right) = 0.9183$

$-\frac{6}{8}\log_2\left(\frac{6}{8}\right) - \frac{2}{8}\log_2\left(\frac{2}{8}\right) = 0.8113$

1 Classification error $= 1 - \max[p(i|t)]$

$= 0.9183 - 1 * \frac{4}{12} - 0.8113 * \frac{8}{12}$

$= 0.0441 > 0$, further split allow



Gini

$G(t) = 1 - \sum_i p(i|t)^2$

Decision trees – high variance – overfit

– handle any data – binary, ordinal, category, numerical

– no scaling, no encoding

Regressor

dtc = DecisionTreeClassifier(criterion = 'Gini', max_features = p, max_depth = 5)

dtc.fit(,)

dtc.predict( )

dtc.tree_.node_count, dtc.tree_.max_depth

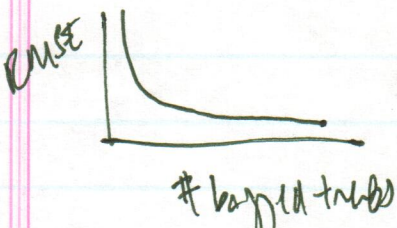
```
param_grid = {'max_depth': range(1, dtc.tree_.max_depth+1, 2),
              'max_features': range(1, len(dtc.feature_importances_)+1, 2)}
GR = GridSearchCV(DecisionTreeClassifier(random_state=42),
                  param_grid=param_grid,
                  scoring='accuracy',
                  n_jobs=-1)
GR.fit(X_train, Y_train)
GR.best_estimator_.tree_.node_count
                  .max_depth
```

'neg_mean_squared_error'

```
export_graphviz(dtc, out_file=dot_data, filled=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('x.png')
```

Ensemble:

bagging = bootstrap aggregating



more trees, less overfit
bagging trees compute in parallel.
bagged variance = $\frac{\sigma^2}{n}$ if trees independent

```
BC = BaggingClassifier(n_estimators=n)   # Regressor
BC.fit(  ,  )
BC.predict(  )
```

Random forest:

Random subset of features ← classification: $\sqrt{m}$
                          ← regression: $m/3$
+ bootstrapping

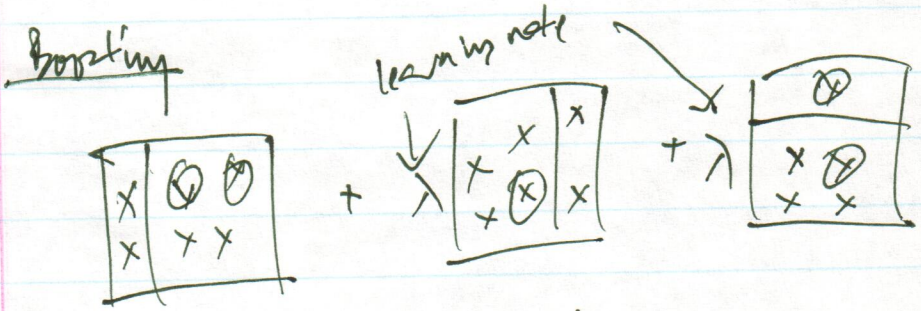
Regressor
rf = RandomForestClassifier(n_estimators = n)
rf.fit(,)
rf.predict( )

Extra Random Trees:
Features + data randomly split — not greedy split.

Regressor
etc = ExtraTreesClassifier(n_estimators = n)
etc.fit(,)
etc.predict( )

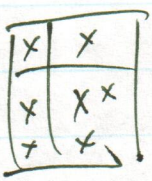
rf.predict_proba( ) → roc_auc_score
rf.feature_importances_

Boosting



3 weak learners

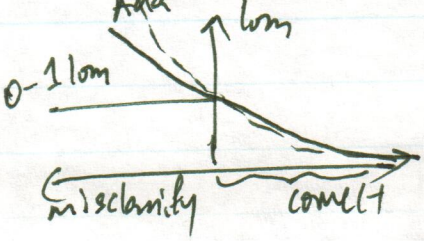
- increase weights of wrong
- decrease weights of right



more complex

more trees with lower learning rate: more regularized
higher learning rate leads to overfitting, less regularized

AdaBoost:



$\text{Log} = e^{-\text{margin}}$

$\log(1+e^{-\text{margin}})$

Bagging

- bootstrap samples
- trained independently /parallel
- only data points considered
- use weights
- will not overfit

Boosting

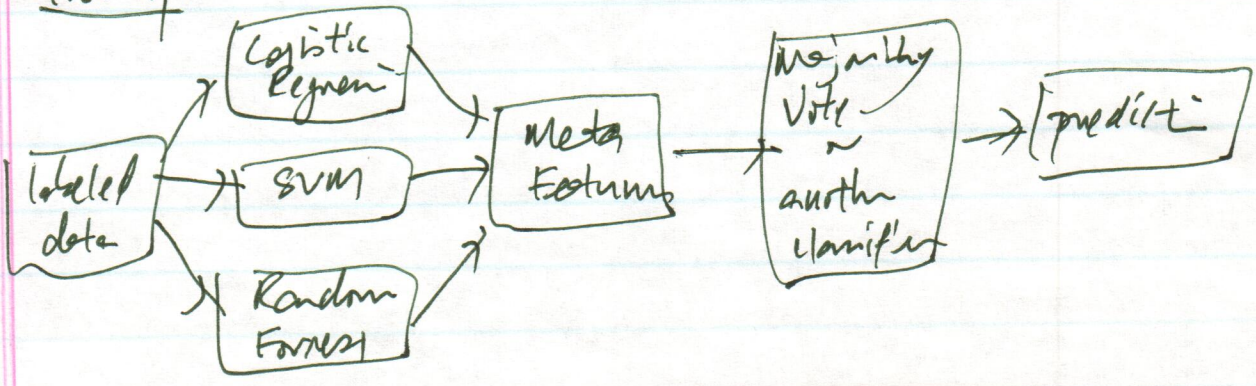
- entire dataset
- created sequentially
- residuals from previous models
- up-weight misclassified points
- may overfit

Tune GB model
- $\lambda < 1$ for regularization
- subsample < 1.0 for fraction of data - SGD
- max_features - limit features

```
gbc = GradientBoostingRegressor(lr = 0.1, max_features = 1,
                                subsample = 0.5, n_estimators = 200)
gbc.fit( , )
gbc.predict( )
```

```
abc = AdaBoostRegressor(base_estimator = DecisionTreeClassifier(),
                        lr = 0.1, n_estimators = n)
```

Stacking:



Base learner

list of fitted models

class based on proba / instead of majority vote

Regressor

```
vc = VotingClassifier(estimator_model_list, voting='soft')
vc.fit(,)
vc.predict()
```

StackingClassifier / Regressor(estimator_list, final_estimator = LogisticRegression())

Imbalanced Dataset:

Downsample - add more importance of minority
- ↑ recall, ↓ precision
- drop majority data.

Upsample - ↑ recall, ↓ precision
- fit duplicate minority class, - Oversample

Random - good
- categorical

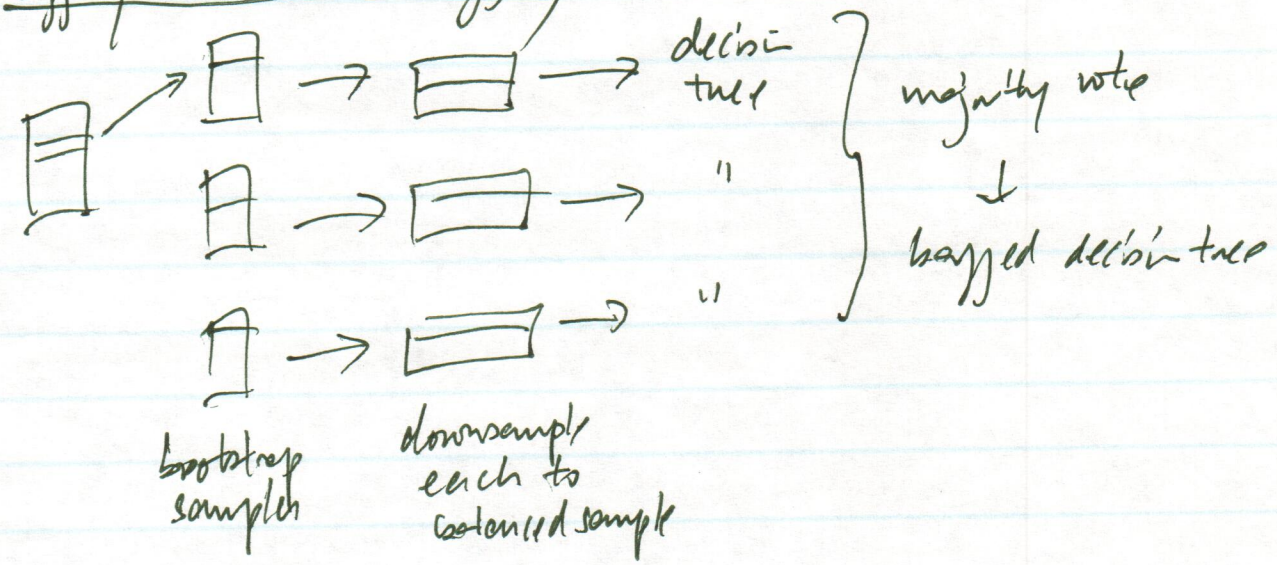
Resample - stratifiedsplit - train test split
ShuffleSplit
Kfold → StratifiedFold

Synetic Oversampling
- choose K nearest neighbor in minority class
- SMOTE - regular
  - borderline - outlier, safe, in-danger
    1- connects in-danger to minority
    2- connect in-danger to whatever is nearby
  - SVM - create new points from borders.
- ADASYN - more weight on values that might be misclassified.
  by add more points around from minority area.

Undersampling -

- Near miss 1 - drop majority points closest to minority
  - thrown off by outliers
- Near Miss 2 - drop closest distance points in minority
  3 -
- Tomek links
- Edited Nearest neighbor - run kNN, remove points don't agree.

Bagging - Balanced Bagging:



Always do train-test split first before under/oversample
Use sensible metric - F1, AUC, Cohen's Kappa

Unsupervised

Clustering

- K-Means
- Hierarchical Agglomerative clustering
- DBSCAN
- mean shift

Dimensionality Reduce

- PCA
- Non-negative matrix factorization (NMF)

		Tabular
ANN	Multi-Layer Perceptron	Predictive problems
RNN	RNN, LSTM	model sequences, forecasting
CNN	CNN	object recognition, segmentation in video/image
Unsupervised	Autoencoders, GANs	Generate images, labelling, dimension reduction.

Gradient Descent:

$-\nabla J(\beta_0, \beta_1)$ → gradient of greatest decrease

$= \frac{\partial J}{\partial \beta_0}, \frac{\partial J}{\partial \beta_1}$

$$w_1 = w_0 - \alpha \nabla \underbrace{\frac{1}{2N}\sum_i^N \left((\beta_0 + \beta_1 x_{obs}^{(i)}) - y_{obs}^{(i)}\right)^2}_{\text{cost function} = J}$$

w_1: next, w_0: current, α: learning rate

until convergence / global minimum.

Gradient $\nabla J = \frac{\partial J}{\partial \beta}$

$= \frac{1}{N}\sum^N \left((\beta_0 + \beta_0 x_{obs}^{(i)}) - y_{obs}^{(i)}\right) x_{obs}^{(i)}$

Stochastic Gradient Descent:

use only a single point

$$w_1 = w_0 - \alpha \nabla \frac{1}{2}\left((\beta_0 + \beta_1 x_{obs}^{(i)}) - y_{obs}^{(i)}\right)^2$$

more noise as only 1 point.

mini-batch:

$n = 1$ to N

$$w_1 = w_0 - \alpha \nabla \frac{1}{2n} \sum_i^n \left(\left(\beta_0 + \beta_1 x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2$$

- reduce memory
- less noisy than SGD

SGD loop

```
for i in range(iter):
    ŷ = np.dot(θ^T, X^T)
    loss_vec = np.sum((y - ŷ)^2)
    grad_vec = (y - ŷ).dot(X) / n
    θ = θ + lr * grad_vec
```

$$h(\theta) = \theta_0 + \theta_1 X$$

cost function: $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h(\theta)^i - y^i \right)^2$

gradient: $\nabla J = \frac{\partial J}{\partial \theta} = \frac{1}{m} \sum_{i=1}^{m} \left(h(\theta)^i - y^i \right) . X_j^i$

update: $\theta_j = \theta_{j-1} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h(\theta)^i - y^i \right) . X_j^i$

```
def cost(θ, X, y)
    c = 1/len(y) * np.sum(np.square((X.dot(θ)) - y))
return c
```

```
def gradient_descent(X, y, θ, α, iterations)
    initialize: θ = np.zeros((iterations, 2))
                costs = np.zeros(iterations)
                m = len(y)

    for i in range(iterations):
        θ = θ - 1/m * α * (X.T.dot((X.dot(θ)) - y))

        θ[i,:] = θ.T
        costs[i] = cost(θ, X, y)

    return θ, θ[ ], costs
```

Backpropagation:

imagine neural net as a function. $F: X \to Y$

F computes many weights

loss function is $J(y, F(x))$

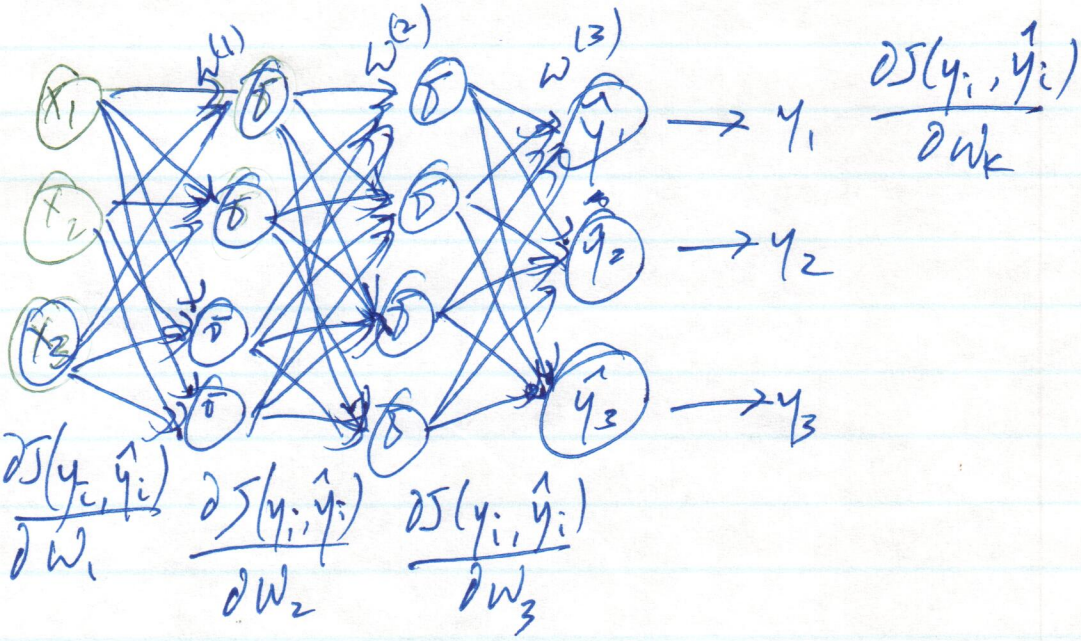
$$F = \begin{cases} z_n = x_1 w_1 + x_2 w_2 + \dots x_n w_n + b \end{cases}$$

calculate $\frac{\partial J}{\partial w_k}$, update w_k

$$\frac{\partial J}{\partial w^{(3)}} = (\hat{y} - y) \cdot a^{(2)} \qquad \frac{\partial J}{\partial w^{(2)}} = \frac{\partial J}{\partial w^3} \cdot a^{2} \qquad \frac{\partial a^2 z^2}{\partial w} = \sigma(z^2)(1-\sigma(z^2)) = \sigma'(z^2)$$

$$\frac{\partial J}{\partial w^{(2)}} = (\hat{y} - y) \cdot \dots \cdot a^{(1)}$$

$$\frac{\partial J}{\partial w^{(1)}} = (\hat{y} - y) \cdot w^{(3)} \cdot \sigma'(z^{(3)}) \cdot w^{(2)} \cdot \dots \cdot x$$



$\sigma'(z) = \sigma(z)(1-\sigma(z)) \leq 0.25$ for sigmoid, max is 0 to 1, so max $0.5(1-0.5) = 0.25$

more layers, gradient gets smaller, vanishing gradient.

$x \xrightarrow{W_1} \underset{z_1}{} a_1 \xrightarrow{W_2} \underset{z_2}{} \hat{y}$

```
def forward_pass(W1, W2):
    z1 = np.dot(X, W1)
    a1 = σ(z1)
    z2 = np.dot(a1, W2)
    ŷ = σ(z2)
```

$$\frac{\partial J}{\partial z_2} = \hat{y} - y$$

$$\frac{\partial J}{\partial W_2} = np.dot\left(\frac{\partial J}{\partial z_2}, a_1\right) = (\hat{y} - y)\, a_1$$

$$\frac{\partial a_1}{\partial z_1} = \sigma'(z_1) = \sigma(z_1)(1 - \sigma(z_1))$$

$$\frac{\partial J}{\partial W_1} = np.dot\left(\frac{\partial J}{\partial W_2}, W_1\right) \sigma'(z_1)\, X = (\hat{y} - y)\, a_1\, W_1\, \sigma'(z_1)\, X$$

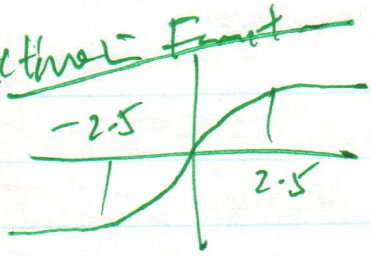
return $\left(\frac{\partial J}{\partial W_1}, \frac{\partial J}{\partial W_2}\right), \hat{y}$

```
def loss_fn(ŷ, y_true, eps = 1e-16):
    loss = -1/N ( y_true log(ŷ) + (1 - y_true) log(1 - ŷ) )
```

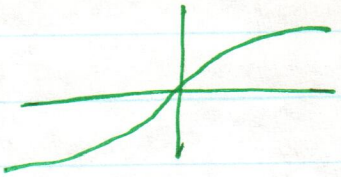
initial W_1, W_2

```
for i in range(num_iter):
    ŷ, (∂J/∂W1, ∂J/∂W2) = forward_pass(W1, W2)
    W1 = W1 - lr ∂J/∂W1
    W2 = W2 - lr ∂J/∂W2
    loss = loss_fn(ŷ, y)
    acc = np.sum(ŷ > 0.5 == y) / N
```

Activation Functions

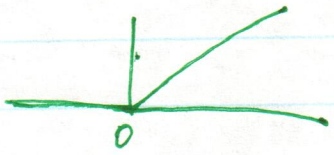


tanh

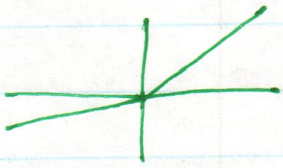


sigmoid $\frac{1}{1+e^{-z}}$

$$\text{ReLU}(z) = \begin{cases} 0 & z<0 \\ z & z \geq 0 \end{cases}$$



$$\text{Leaky ReLU}(z) = \begin{cases} \alpha z, & z<0 \\ z, & z \geq 0 \end{cases}$$



Regularization

ridge

- regularization penalty in loss function - $J = \frac{1}{2n}\sum_i^n (\hat{y}-y)^2 + \lambda \sum_i w_i^2$
- dropout - randomly remove subsets of neurons.
- early stopping - no change in val loss after patience of n
- stochastic / mini-batch GD

Optimizers

$w := w - \alpha \nabla J$

momentum - $v_t = \eta v_{t-1} - \alpha \cdot \nabla J$

(larger steps if moving in ~~large~~ same direction. $w := w - v_t$

Nesterov momentum -

Adagrad - scale update for each weight

$$w := w - \frac{\eta}{\sqrt{G}} \nabla J$$

RMSProp – decay older gradients more than recents.

Adam – combines momentum + RMSProp

Epoch – one pass through all training data.

Full batch GD – 1 step per epoch
SGD – n steps per epoch (n = training size)
minibatch – $\frac{n}{\text{batchsize}}$ steps per epoch

Keras:

```
model = Sequential()
model.add(Dense(units=4, input_dim=3, input_shape=(3,)))
model.add(Activation('sigmoid'))
model.add(Dense(units=4))
model.add(Activation('sigmoid', 'relu', 'softmax'))
model.summary()
model.compile(SGD(lr= ), optimizer= , 'binary_crossentropy', loss=
              metric=['accuracy'])
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=n, shuffle=True)
y_pred_class = model.predict_classes(X_test)
y_pred_proba = model.predict(X_test)
~~y_pred_class = model.np.argmax(y_pred_class, axis=1)~~
```

Multiclass classification:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

$$\text{categorical cross entropy} = -\sum_i y_i \log(\hat{y}_i)$$

$$\frac{\partial CE}{\partial \text{softmax}} \cdot \frac{\partial \text{softmax}}{\partial z_i} = \hat{y}_i - y_i$$

Scaling:

minmax (0, 1): $x_i = \frac{X_i - X_{min}}{X_{max} - X_{min}}$

linear scaling (-1, 1): $X_i = 2\left(\frac{X_i - X_{min}}{X_{max} - X_{min}}\right) - 1$

Convolution:

Padding
Stride
Convolution size

$$\text{output size} = \frac{(\text{input size} + 2 * \text{padding} - (\text{kernel size} - 1) - 1)}{\text{stride}} + 1$$

Pooling - reduce image size
- max pooling, avg pooling

Transfer Learning

Convolution → Fully Connected layers → Predict
→ new layers → new predict

```
if K.image_data_format() == 'channel_first':
    input_shape = (1, img_rows, img_cols)
else:
    input_shape = (img_rows, img_cols, 1)
```

```
freeze layers:
    for l in feature_layers:
        l.trainable = False
```

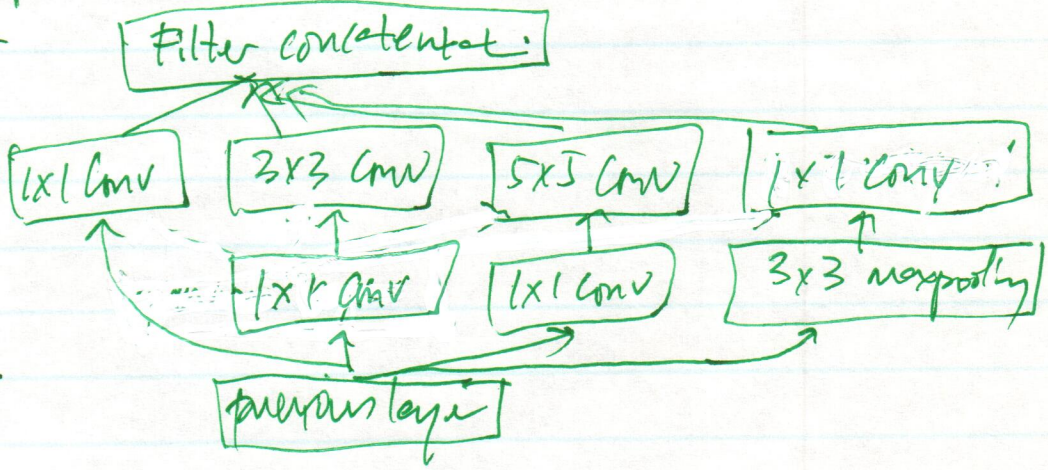
CNN:

LeNet

AlexNet - data augmentation - crop, flipping, zoom

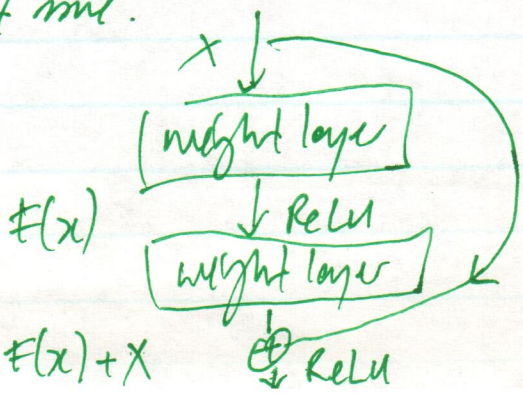
VGG - deep network w 3x3 convolutions.

Inception -



1x1 conv reduces depth.

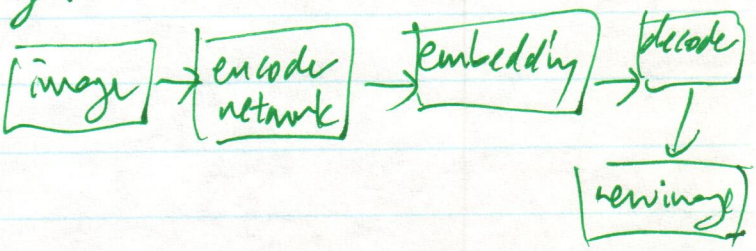
ResNet - early layers slow to adjust due to vanishing gradient issue.



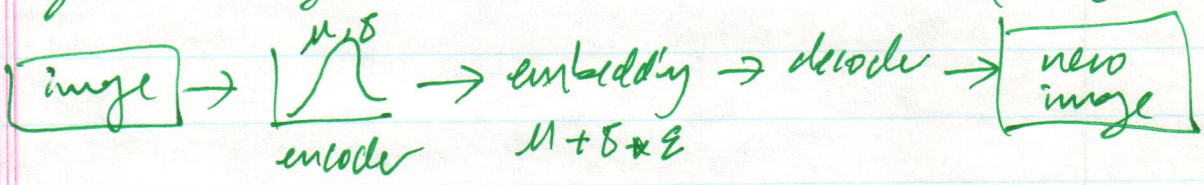
RNN:
Forecasting, speech, genome,

LSTM:
Forecasting, speech, anomaly detection

Autoencoders:
- learn similarity/difference in images.
- reduce dimensionality of images.
- anomaly detection
- drug discovery ....

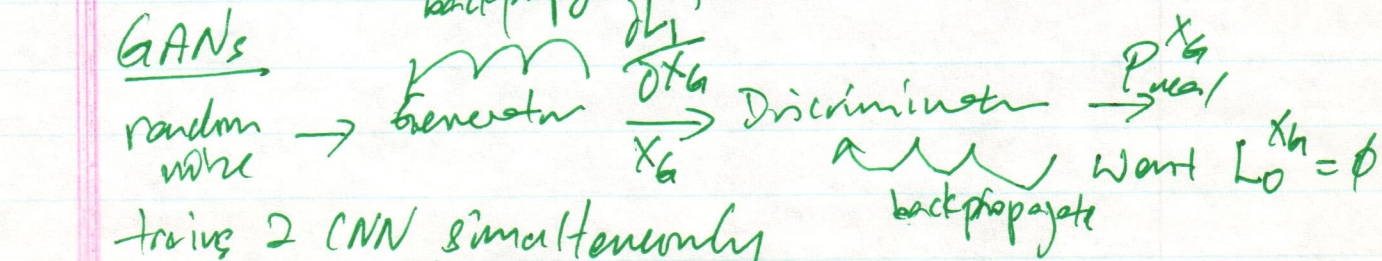


Variational AE:
generate images based on normal distribution of images.



penalty for not reconstructing new images correctly
generating μ, σ equal to 0 and 1
KL Divergence = $\frac{1}{2}(e^{\log(\sigma)}(\log(\sigma)+1) + \mu^2)$
penalizes for $\log(\sigma)$, μ different from ϕ.

GANs



training 2 CNN simultaneously
$P_{real}^{X_G}$ - probability generated image is real
compute loss assuming $P_{real}^{X_G} = \phi$ (false)

$L_0^{x_G} = f(P_{real}^{x_G}, \phi)$ $L_1^{x_G} = f(P_{real}^{x_G}, 1)$

More sensitive to hyperparameters than other DL models.

LIME (Locally-Interpretable Model Agnostic Explanations):
- sensitivity analysis of outputs to inputs.

Reinforcement Learning
Agents interact in Environment
- choice of Actions in response to State limited by Policy
- imparts by Rewards, not known.