

# Udacity: Intro to ML with PyTorch

## Contents

<b>1</b>	<b>Supervised Learning</b>	<b>1</b>
1.1	Linear Regression (doc)	1
1.2	Perceptron Algorithm	3
1.3	Decision Trees (doc)	3
1.4	Naive Bayes (doc)	5
1.5	Support Vector Machines (SVMs) (doc)	6
1.6	Ensemble Methods (doc)	7
1.7	Model Evaluation Metrics	8
	1.7.1 Classification Metrics (formulas)	8
	1.7.2 Regression Metrics	9
1.8	Training and Tuning	10
<b>2</b>	<b>Deep Learning</b>	<b>12</b>
2.1	Neural Networks	12

# 1 Supervised Learning

## 1.1 Linear Regression (doc)

**Absolute Trick:** A point at  $(p, q)$  with a line  $y = w_1x + w_2$ , we want to change the line to slowly move it towards the point. We want to do this by a small value  $\alpha$  (learning rate), so our line become  $\hat{y} = (w_1 + p\alpha)x + (w_2 + \alpha)$ .

**Square Trick:** Similar to the absolute trick, but we consider the offset of the point from the y-axis ( $p$ ) and from the line ( $q - q'$ ) where  $q'$  is the *estimated*  $y$  point and  $q$  is the actual  $y$  point. So our line becomes  $\hat{y} = (w_1 + p(q - q')\alpha)x + (w_2 + (q - q')\alpha)$

**Mean Absolute Error:** the average of the sum of the distance between the estimate line value  $\hat{y}$  and actual  $y$  value, giving us the equation:  $Error = \frac{1}{m} \sum_{i=1}^m |y - \hat{y}|$

**Mean Squared Error:** Similar to MAE, but we square the value instead which gives us a square near the value  $y - \hat{y}$ . This gives us the equation:  $Error = \frac{1}{2m} \sum_{i=1}^m (y - \hat{y})^2$

**Gradient Step:**  $w_i \rightarrow w_i - \alpha \frac{\partial}{\partial w_i} Error$  with  $\frac{\partial}{\partial w_1} Error = -(y - \hat{y})x$  and  $\frac{\partial}{\partial w_2} Error = -(y - \hat{y})$

```
1 X : array of predictor features
2 y : array of outcome values
3 W : predictor feature coefficients
4 b : regression function intercept
5 learn_rate : learning rate
6
7 yhat = np.matmul(X,W) + b # predicted values
8 error = y-yhat
9 W_new = W + learn_rate*np.matmul(error,X) # updated slope
10 b_new = b + learn_rate*error.sum() # updated y-intercept
```

**Higher Dimensions:**  $\hat{y} = w_1x_1 + w_2x_2 + \dots + w_{n-1}x_{n-1} + w_n$

**Polynomial Regression:** Allows us to add degrees to our  $x$  to better find our line.

```
1 import numpy as np
2 import pandas as pd
3 from sklearn.preprocessing import PolynomialFeatures
4 from sklearn.linear_model import LinearRegression
5
6 train_data = pd.read_csv('data.csv')
7 X = np.asarray(train_data['Var_X']).reshape((20,1))
8 y = np.asarray(train_data['Var_Y']).reshape((20,1))
9
10 poly_feat = PolynomialFeatures(degree=4) # 4th degree polynomial
11 X_poly = poly_feat.fit_transform(X)
12
13 poly_model = LinearRegression(fit_intercept=False).fit(X_poly, y)
```

**L1 Regularization:** adds the absolute value of the coefficients ( $w_i$ ) to the error. This is computationally inefficient, better when data is sparse, gives us feature selection (makes irrelevant columns to 0).

**L2 Regularization:** we add the square of the coefficients ( $w_i$ ) to the error. This is computationally efficient, better for non-sparse data, doesn't have feature selection (treats all columns equally).

$\lambda$  **Parameter:** determines how much impact the coefficients have on our total error. Choosing a large  $\lambda$  leads to simpler models more while a small  $\lambda$  leads to more complex models.

We can train a Linear Regression model with L1 regularization applied to it by using the [Lasso model](#).

```
1 import pandas as pd
2 from sklearn.linear_model import Lasso
3
4 train_data = pd.read_csv('data.csv', header=None)
5 X = train_data.iloc[:, :-1] # all rows, all columns up to last
6 y = train_data.iloc[:, -1] # all rows, last column
7
8 lasso_reg = Lasso().fit(X, y)
9
10 reg_coef = lasso_reg.coef_
11 print(reg_coef)
12 # [ 0.  2.35793224  2.00441646 -0.05511954 -3.92808318  0.]
```

**Feature Scaling:** a way of transforming your data into a common range of values (2 ways below).

**Standardizing:** turning the column into a standard normal variable by computing  $\frac{x-\mu}{\sigma}$  and is interpreted as the number of standard deviations each value is away from the mean (most common type).

**Normalizing:** data is scaled between 0 and 1 by calculating  $\frac{x-x.min}{x.max-x.min}$

**Distance Based Metrics:** If an algorithm uses distance based metrics to predict, choosing some sort of feature scaling is necessary (such as for SVMs or k-nn).

**When using Regularization:** we need all columns to have equal ranges so they are treated similar when calculating the penalty. We need features with small ranges and large ranges to have similar coefficients.

```
1 import pandas as pd
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.linear_model import Lasso
4
5 train_data = pd.read_csv('data.csv', header=None)
6 X = train_data.iloc[:, :-1]
7 y = train_data.iloc[:, -1]
8
9 scaler = StandardScaler()
10 X_scaled = scaler.fit_transform(X)
11
12
13 lasso_reg = Lasso().fit(X_scaled, y)
14
15 reg_coef = lasso_reg.coef_
16 print(reg_coef)
17 # [0.  3.90753617  9.02575748 -0.  -11.78303187  0.45340137]
```

Notice how after we scaled the features, the L1 regularization set the 1st and 4th columns coefficients to zero. Where when we did not standardize in the previous code, the L1 regularization set the 1st and 6th columns coefficients to zero.

## 1.2 Perceptron Algorithm

**Linear Boundaries:** we use a boundary line to determine how to classify the data, denoted by the equation  $w_1x_1 + w_2x_2 + b = 0$  which we rewrite as  $Wx + b = 0$  where  $W = (w_1, w_2)$  and  $x = (x_1, x_2)$ . Note that we just take the dot product of  $W \cdot x \implies w_1x_1 + w_2x_2$ . We then classify the output of the boundary line by:

$$\hat{y} = \begin{cases} 1 & \text{if } Wx + b \geq 0 \\ 0 & \text{if } Wx + b < 0 \end{cases}$$

**Higher Dimensions:** if we are working with  $n$  columns, our boundary line becomes a  $n - 1$  dimensional hyperplane with the equation  $w_1x_1 + w_2x_2 + \dots + w_nx_n + b = 0 \implies Wx + b = 0$ . The prediction works the same way as the linear boundary. Note that weights ( $W$ ) is a  $(1 \times n)$  row vector, input ( $x$ ) is a  $(n \times 1)$  column vector, and bias ( $b$ ) is a  $(1 \times 1)$  vector.

**Basic Perceptrons:** We have our input nodes ( $x_1, \dots, x_n$  and can also include our bias  $b$ ) which are then multiplied by the corresponding weights ( $w_1, \dots, w_n$ ), which are then inputted into a linear function node ( $Wx + b$ ), and finally that input is sent to a step function node (determines if  $\hat{y}$  is 1 or 0).

**Updating the line:** In a 2D example, with point  $(p, q)$ , we can update the boundary line with the equation  $(w_1 \pm \alpha p)x_1 + (w_2 \pm \alpha q)x_2 + (b \pm \alpha * 1)$ . Note that  $\pm$  depends on which way we move the line.

```
1 def perceptronStep(X, y, W, b, learn_rate = 0.01):
2     for i in range(len(X)): # go through all points
3         yhat = prediction(X[i], W, b) # get predicted values
4         if y[i] - yhat == 1: # if y=1 but yhat=0 (move line down)
5             W[0] += learn_rate*X[i][0]
6             W[1] += learn_rate*X[i][1]
7             b += learn_rate
8         if y[i] - yhat == -1: # if y=0 but yhat=1 (move line up)
9             W[0] -= learn_rate*X[i][0]
10            W[1] -= learn_rate*X[i][1]
11            b -= learn_rate
12    return W, b
```

## 1.3 Decision Trees (doc)

**(Multiclass) Entropy:**  $-\sum_{i=1}^n p_i \log_2(p_i)$  where  $p_i$  is the probability for each class. Note that points with the same color will have zero entropy, so this is the ideal outcome for classification.

**Information Gain:** Tells us how well our data is being classified (in a range between 0 and 1). The lower the information gain, the worse our model is doing as separating the data. The higher the information gain, the better our model is performing at classification. We have the formula:

$$IG = Entropy(Parent) - \left[ \frac{m}{m+n} Entropy(Child_1) + \frac{n}{m+n} Entropy(Child_2) \right]$$

**Important Hyperparameters for Decision Trees:**

**Maximum depth:** the largest possible length between the root to a leaf, a tree of maximum length  $k$  can have at most  $2^k$  leaves.

**Minimum samples to split:** numeric value ( $m$ ) that says if a node doesn't have at least  $m$  samples it can't split.

**Minimum samples per leaf:** decides how many samples must be in each node. If it is an integer, then it is a sample size. If it is a float, it is a percentage of samples needed in a leaf.

**WARNING:** Large depth often causes overfitting. Small depth can result in a very simple model, which may cause underfitting. Small minimum samples per split may result in overfitting. Large minimum samples may result in the tree not having enough flexibility to get built, and may result in underfitting.

```

1  import pandas as pd
2  import numpy as np
3  from sklearn.model_selection import train_test_split
4  from sklearn.tree import DecisionTreeClassifier
5  from sklearn.metrics import accuracy_score
6  import random
7  random.seed(42)
8
9  in_file = 'titanic_data.csv'
10 full_data = pd.read_csv(in_file)
11
12 outcomes = full_data['Survived'] # Labels
13 features_raw = full_data.drop('Survived', axis = 1) # Remove label from features df
14
15 features_no_names = features_raw.drop(['Name'], axis=1) # Remove names
16
17 features = pd.get_dummies(features_no_names) # One-hot encoding
18 features = features.fillna(0.0)
19
20 len(features.columns) # 839
21 # Note that we now have 839 columns instead of 11 since each ticket is a column
22 # from the one-hot encoding (each ticket_number is now a column with all 0's
23 # expect for the person/row with the ticket, which will be a 1).
24
25 X_train, X_test, y_train, y_test = train_test_split(features, outcomes,
26                                                    test_size=0.2, random_state=42)
27
28 model = DecisionTreeClassifier().fit(X_train, y_train)
29
30 # Making predictions
31 y_train_pred = model.predict(X_train)
32 y_test_pred = model.predict(X_test)
33 train_accuracy = accuracy_score(y_train, y_train_pred) # 1.0
34 test_accuracy = accuracy_score(y_test, y_test_pred) # 0.8156
35 # High training and lower test accuracy means we might be overfitting.
36
37 # Tune the hyperparameters for new model to improve test score
38 model_tune = DecisionTreeClassifier(max_depth=20, min_samples_leaf=6)
39 model_tune.fit(X_train, y_train)
40 tune_y_train_pred = model_tune.predict(X_train)
41 tune_y_test_pred = model_tune.predict(X_test)
42 tune_train_accuracy = accuracy_score(y_train, tune_y_train_pred)
43 tune_test_accuracy = accuracy_score(y_test, tune_y_test_pred)
44
45 print('Train Acc: {}, Test Acc: {}'.format(tune_train_accuracy, tune_test_accuracy))
46 # Train Acc: 0.8820224719101124, Test Acc: 0.8603351955307262

```

## 1.4 Naive Bayes (doc)

**Naive Bayes:** a supervised machine learning algorithm that can be trained to classify data into multi-class categories. The probabilistic model computes the conditional probabilities of the input features and assigns the probability distributions to each of possible classes.

**Bayes Theorem:** We have prior probabilities for a given problem, and after we know that event  $B$  occurred, we can find the posterior probability with the formula  $P(A_j|B) = \frac{P(B|A_j)P(A_j)}{\sum P(B|A_i)P(A_i)}$

**Naive Assumption:**  $P(A \cap B) = P(A)P(B)$  meaning that the two events are independent. We assume this to be true even if it is not the case.

**Conditional Probability (Proportional):**  $P(A|B_1, B_2, \dots, B_n) \propto P(B_1, B_2, \dots, B_n|A)P(A)$

**Algorithm:** For an event  $A$ , we have  $P(A|B_1, B_2, \dots, B_n) \propto \frac{P(B_1|A)P(B_2|A)\dots P(B_n|A)P(A)}{P(B_1, B_2, \dots, B_n)}$

**Bag of Words (BoW):** a collection of text data where we count the frequency of the words.

**Accuracy:** the ratio of the number of correct predictions to the total number of predictions.

**Precision:** a ratio of true positives to all positives (true+false positive).

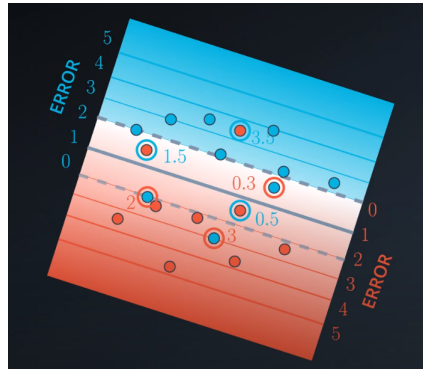
**Recall (Sensitivity):** a ratio of true positives to all the words (true positives + false negatives).

**F1 Score:** is the weighted average of the precision and recall scores (between 0 and 1, higher better).

```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.feature_extraction.text import CountVectorizer
4 from sklearn.naive_bayes import MultinomialNB
5 from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
6
7 # Read in data from table, add column names, specify separator
8 df = pd.read_table('smsspamcollection/SMSSpamCollection', sep='\t',
9                   names=['label', 'sms_message'])
10
11 # Convert labels to numerical encodings
12 df['label'] = df.label.map({'ham':0, 'spam':1})
13
14 X_train, X_test, y_train, y_test = train_test_split(df['sms_message'], df['label'],
15                                                    random_state=1)
16
17 count_vector = CountVectorizer()
18 # Note that CountVectorizer automatically converts to lowercase and ignores
19 # all punctuation. We can add stop_words=English if we want to ignore
20 # common english words (a, an, and) in our matrix.
21
22 # Fit the training data and return the frequency matrix
23 training_data = count_vector.fit_transform(X_train)
24
25 # Transform testing data and return the matrix (don't fit testing data)
26 testing_data = count_vector.transform(X_test)
27
28 naive_bayes = MultinomialNB()
29 naive_bayes.fit(training_data, y_train)
30 predictions = naive_bayes.predict(testing_data)
31
32 print('Accuracy score: ', format(accuracy_score(y_test, predictions))) # 0.98851
33 print('Precision score: ', format(precision_score(y_test, predictions))) # 0.972067
34 print('Recall score: ', format(recall_score(y_test, predictions))) # 0.94054
35 print('F1 score: ', format(f1_score(y_test, predictions))) # 0.95604
```

## 1.5 Support Vector Machines (SVMs) (doc)

**Classification Error:** we have a line separating our data points into groups,  $Wx + b = 0$ , and we want to add two more lines called the *margin*. The line above has equation  $Wx + b = 1$  and the line below has equation  $Wx + b = -1$ . We then assign points error based on where they are in respect to the margin:



**Classification Error:** we take the absolute value of all misclassified points and add them together.

**Margin:** we want a small error if the margin is large, and a large error if the margin is small. We can determine the size of the margin by  $\frac{2}{|W|}$  (2 divided by the norm of  $W$ ).

**Margin Error:**  $|W|^2$  (note this is the same value given by L2 regularization).

**Error Function:** we want  $\text{Error} = \text{Classification Error} + \text{Margin Error}$ , which we will minimize using gradient descent.

**The C Parameter:**  $\text{Error} = C * \text{Classification Error} + \text{Margin Error}$ . It is a constant we use to determine the influence of the classification error. Large  $C$  means focus on classifying points (smaller margin), small  $C$  means focus on margin error (makes more classification errors).

**Linear Kernel:** we separate our data using a linear line (doesn't work well when data is complicated).

**Polynomial Kernel:** we separate our data using circles, hyperbolas, parabolas, and many more boundaries by adding more dimensions to our data to classify it. The degree of the polynomial determines what kind of boundaries we can create.

ex: (degree 2)  $\mathbb{E}^2 \rightarrow \mathbb{E}^5 : (x, y) \rightarrow (x, y, x^2, xy, y^2)$ . We map our points  $(x, y)$  to a higher dimensions which allows us to create degree 2 functions. We then find a boundary hyperplane and map this back to a 2nd degree polynomial boundary (think of a paraboloid in 3D mapping back to a circle in 2D).

**RBF Kernel:** We build mountain ranges (radial basis functions) on top of our point to find boundaries to group the data. The plane that intersects the paraboloids is the projection that determine the boundaries. The  $\gamma$  hyperparameter can be tuned to determine the width of the curve, with a small  $\gamma$  giving us a wide curve and a large  $\gamma$  giving us a narrow curve.

```
1 from sklearn.svm import SVC
2 from sklearn.metrics import accuracy_score
3
4 data = np.asarray(pd.read_csv('data.csv', header=None))
5 X = data[:,0:2] # features
6 y = data[:,2] # labels
7
8 model = SVC(kernel='rbf', gamma=27).fit(X,y)
9
10 y_pred = model.predict(X)
11 acc = accuracy_score(y, y_pred)
12
```

## 1.6 Ensemble Methods (doc)

**Bias:** when a model has high bias, this means that it doesn't do a good job of bending to the data. One example is linear regression, which fits a straight line no matter how the data is shaped (but has a low variance).

**Variance:** when a model has high variance, this means that it changes drastically to meet the needs of every point in our dataset. One example are decision trees, which will attempt to split every point into its own branch if possible (but has low bias).

**High Bias, Low Variance:** models tend to underfit data, as they are not flexible (linear models).

**High Variance, Low Bias:** models tend to overfit data, as they are too flexible (decision trees).

**Randomness:** introduction of randomness to high variance algorithms to combat the tendency of these algorithms to overfit the data.

**Bootstrap:** sampling the data with replacement and fitting your algorithm to the sampled data.

**Boosting:** Random sampling with replacement over weighted data.

**Random Forests:** take a subset of the columns and build a decision tree from them, then repeat this process multiple times. We then use each tree to predict, and pick the most occurring outcome.

**Bagging:** take subsets of the data, create one-node decision trees to find a boundary for each subset, then take the overlap of the boundaries to determine the final boundary line (bootstrapping).

**AdaBoost:** we assign weights to each points, and we want to minimize the sum of the weights on the incorrectly classified points. We then increase the weights (by  $\frac{\sum \text{correct weights}}{\sum \text{incorrect weights}}$ ) of the misclassified points and create a new boundary line. We continue this process for multiple models.

Once we have our models, we assign weight to the models to determine the influence they have in our final boundary, which can be calculated by  $weight = \ln(\frac{\# \text{correct}}{\# \text{incorrect}})$

```
1  # NOTE: we are using the testing and training data from the Naive Bayes
2  # section, so refer to that to see data preprocessing steps
3
4  from sklearn.ensemble import AdaBoostClassifier
5  from sklearn.ensemble import BaggingClassifier
6  from sklearn.ensemble import RandomForestClassifier
7  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
8
9  bag_model = BaggingClassifier(n_estimators=200).fit(training_data, y_train)
10 rf_model = RandomForestClassifier(n_estimators=200).fit(training_data, y_train)
11 ada_model = AdaBoostClassifier(n_estimators=300, learning_rate=0.2)
12 ada_model.fit(training_data, y_train)
13
14 bag_ypred = bag_model.predict(testing_data)
15 rf_ypred = rf_model.predict(testing_data)
16 ada_ypred = ada_model.predict(testing_data)
17
18 # General Format for printing evaluation metrics (scoring model)
19 print('Accuracy score: ', format(accuracy_score(y_true, preds)))
20 print('Precision score: ', format(precision_score(y_true, preds)))
21 print('Recall score: ', format(recall_score(y_true, preds)))
22 print('F1 score: ', format(f1_score(y_true, preds)))
23
```

NOTE: the *base\_estimator* parameter for AdaBoost and Bagging allows you to choose the weak learner model to be used in the classification process.



## 1.7 Model Evaluation Metrics

### 1.7.1 Classification Metrics ([formulas](#))

**Confusion Matrix:** a table that tells us the True Positive, False Negatives, False Positive, and True Negatives (left to right, top to bottom).

**Error Types:** False Positives are classified as *Type 1 Error*, False Negative are *Type 2 Error*.

**Accuracy:** used to compare models, it tells us the proportion of observations we correctly labeled (don't use if classes are imbalanced).

**Precision:** focuses on *predicted* positive values, helps determine if you are doing a good job of predicting the positive values, as compared to predicting negative values as positive. Avoiding false positives.

**Recall:** focuses on the *actual* positive values, determines if you are doing a good job of predicting the positive values without regard of how you are doing on the actual negative values. Avoiding false negatives.

**F1 Score:** the harmonic mean (less than regular mean), found by  $2 * \frac{Precision * Recall}{Precision + Recall}$  (care equally about both the positive and negative cases).

**$F_\beta$  Score:** If we want our model to focus more on precision than recall (such as spam filtering), we can use a smaller  $\beta$ . If we want our model to focus more on recall than precision (such as medical diagnosis), we can use a larger  $\beta$ . The formula is  $F_\beta = (1 + \beta^2) \frac{precision * recall}{\beta^2 * precision + recall}$

**ROC Curve & AUC:** By finding different thresholds for our classification metrics, we can measure the area under the ROC curve. When the AUC is closer to 1, the better your model is performing.

```
1  # Again, we are using the data from the Naive Bayes section, so refer
2  # back to that to see the data preprocessing steps
3
4  from sklearn.naive_bayes import MultinomialNB
5  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
6  fbeta_score
7  from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,
8  from sklearn.svm import SVC
9
10 naive_bayes = MultinomialNB().fit(training_data, y_train)
11 bag_mod = BaggingClassifier(n_estimators=200).fit(training_data, y_train)
12 rf_mod = RandomForestClassifier(n_estimators=200).fit(training_data, y_train)
13 svm_mod = SVC().fit(training_data, y_train)
14
15 nb_preds = naive_bayes.predict(testing_data)
16 bag_preds = bag_mod.predict(testing_data)
17 rf_preds = rf_mod.predict(testing_data)
18 svm_preds = svm_mod.predict(testing_data)
19
20 def metric_scores(actual, preds, model, model_name):
21     print('Accuracy for ' + model_name + ' : {}'.format(accuracy_score(actual, preds)))
22     print('Precision ' + model_name + ' : {}'.format(precision_score(actual, preds)))
23     print('Recall for ' + model_name + ' : {}'.format(recall_score(actual, preds)))
24     print('F1 for ' + model_name + ' : {}'.format(f1_score(actual, preds)))
25     print('\n')
26     return None
27
28 metric_scores(y_test, nb_preds, naive_bayes, 'Naive Bayes')
29 metric_scores(y_test, bag_preds, bag_mod, 'Bagging')
30 metric_scores(y_test, rf_preds, rf_mod, 'Random Forest')
31 metric_scores(y_test, svm_preds, svm_mod, 'SVM')
32 # Running this, Naive Bayes performed best for all expect precision (rf_model)
33 fbeta_score(y_test, nb_preds, beta=1) # same output as F1 score since beta=1
```

```

1  from itertools import cycle
2  from sklearn.metrics import roc_curve, auc, roc_auc_score
3  from scipy import interp
4
5  def build_roc_auc(model, X_train, X_test, y_train, y_test):
6      y_preds = model.fit(X_train, y_train).predict_proba(X_test)
7      # Compute ROC curve and ROC area for each class
8      fpr = dict()
9      tpr = dict()
10     roc_auc = dict()
11     for i in range(len(y_test)):
12         fpr[i], tpr[i], _ = roc_curve(y_test, y_preds[:, i])
13         roc_auc[i] = auc(fpr[i], tpr[i])
14
15     # Compute micro-average ROC curve and ROC area
16     fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_preds[:, 1].ravel())
17     roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
18
19     plt.plot(fpr[2], tpr[2], color='darkorange', lw=2,
20             label='ROC curve (area = %0.2f)' % roc_auc[2])
21     plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
22     plt.xlim([0.0, 1.0])
23     plt.ylim([0.0, 1.05])
24     plt.xlabel('False Positive Rate')
25     plt.ylabel('True Positive Rate')
26     plt.title('Receiver operating characteristic example')
27     plt.show();
28
29     return roc_auc_score(y_test, np.round(y_preds[:, 1]))

```

## 1.7.2 Regression Metrics

**Mean Absolute Error (MAE):** a useful metric to optimize on when the value you are trying to predict follows a skewed distribution (not as much influence from outliers as MSE). The optimal value for this technique is the median value.

**Mean Squared Error (MSE):** most used metric for optimization (impacted by outliers and skew), it is differentiable and can be used better for gradient descent. The optimal value is the mean.

**R2 Score:** interpreted as the ‘amount of variability’ captured by a model. Minimizing MSE will maximize R2, which is good since we want this score to be as close to 1 as possible.

```

1  # Import all regression models
2  from sklearn.linear_model import LinearRegression
3  from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor
4  from sklearn.tree import DecisionTreeRegressor
5  from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
6
7  def reg_metrics(actual, preds, model, model_name):
8      print('R2 ' + model_name + ' : {}'.format(r2_score(actual, preds)))
9      print('MAE ' + model_name + ' : {}'.format(mean_absolute_error(actual, preds)))
10     print('MSE ' + model_name + ' : {}'.format(mean_squared_error(actual, preds)))
11     print('\n')
12     return None

```

Using the Boston housing dataset, if we were to fit and predict on all of the models above, we would see that the RandomForestRegressor performs the best on all evaluation metrics.

## 1.8 Training and Tuning

**Underfitting:** making our model too simple, leading to bad accuracy on the training and testing set (an error due to high bias). Does not generalize well to new data.

**Overfitting:** making our model too complicated, leading to higher training set accuracy but low testing set accuracy (an error due to high variance). Does not generalize well to new data.

**K-Fold Cross Validation:** splitting our data  $k$  time into training and testing sets to train and evaluate our model on different data combinations, so we don't lose some of the data.

**Learning Curves:** In a model that is underfitting, the training and CV error converge at a high error. In a model that is overfitting, the training and CV error do not converge. In a good model, the training and CV error will converge at a low error point.

**Grid Search:** combines all specified combinations of hyperparameters to find the best model. We then pick the one with the highest F1 score as our model and use a testing set to verify.

```
1  import pandas as pd
2  import numpy as np
3  from sklearn.datasets import load_diabetes
4  from sklearn.model_selection import train_test_split, RandomizedSearchCV
5  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
6  from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
7  import matplotlib.pyplot as plt
8  from sklearn.svm import SVC
9  import seaborn as sns
10 sns.set(style="ticks")
11
12 diabetes = pd.read_csv('diabetes.csv')
13
14 # Data Analysis
15 diabetes.describe() # see summary statistics for each column
16 diabetes.isna().any() # (0) - find number of NA values (by column)
17 diabetes['Outcome'].sum()/len(diabetes) # (0.35) - proportion of diabetes outcomes
18 diabetes['Age'].value_counts().hist() # right-skewed data
19 diabetes['Glucose'].hist() # approx normal distribution
20 sns.heatmap(diabetes.corr(), annot=True, cmap="YlGnBu"); # correlation heatmap
21
22 y = diabetes['Outcome']
23 X = diabetes.drop(columns='Outcome', axis=1)
24 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
25                                                    random_state=42)
26
27 # Build a RandomForestClassifier and use RandomizedSearch to find best params
28 clf_rf = RandomForestClassifier()
29
30 param_dist = {"max_depth": [3, None],
31              "n_estimators": list(range(10, 200)),
32              "max_features": list(range(1, X_test.shape[1]+1)),
33              "min_samples_split": list(range(2, 11)),
34              "min_samples_leaf": list(range(1, 11)),
35              "bootstrap": [True, False],
36              "criterion": ["gini", "entropy"]}
37
38 random_search = RandomizedSearchCV(clf_rf, param_distributions=param_dist)
39 random_search.fit(X_train, y_train)
40
41 rf_preds = random_search.best_estimator_.predict(X_test)
42 print('F1 Score: {}'.format(f1_score(y_test, rf_preds))) # 0.6847
```

```

1 # Build a AdaBoostClassifier and use RandomizedSearch to find best params
2 ada = AdaBoostClassifier()
3
4 parameters = {'n_estimators': list(range(50, 275, 25)),
5               'learning_rate': [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 5, 10, 15]}
6
7 ada_random = RandomizedSearchCV(ada, param_distributions=parameters)
8 ada_random.fit(X_train, y_train)
9
10 ada_preds = ada_random.best_estimator_.predict(X_test)
11 print('F1 Score: {}'.format(f1_score(y_test, ada_preds))) # 0.6729
12
13 # Plotting feature importance
14 features = diabetes.columns[:-1]
15 importances = random_search.best_estimator_.feature_importances_
16 indices = np.argsort(importances)
17
18 plt.title('Feature Importance (Random Forest Classifier)')
19 plt.barh(range(len(indices)), importances[indices], color='b', align='center')
20 plt.yticks(range(len(indices)), [features[i] for i in indices])
21 plt.xlabel('Relative Importance')
22 plt.show() # Glucose, BMI, and Age were the top 3 features

```

Note that these features ranking were very similar to the heatmap that was made (except for pregnancies, which turned out to be the least influential feature).

In this case study, we looked at predicting diabetes for 768 patients. There was a reasonable amount of class imbalance with just under 35% of patients having diabetes. There were no missing data, and initial looks at the data showed it would be difficult to separate patients with diabetes from those that did not have diabetes.

Two advanced modeling techniques were used to predict whether or not a patient has diabetes. The most successful of these techniques proved to be the RandomForestClassifier, which had the following metrics:

Accuracy score for random forest : 0.7727272727272727

Precision score random forest : 0.6785714285714286

Recall score random forest : 0.6909090909090909

F1 score random forest : 0.6846846846846847

Based on the initial look at the data, it is unsurprising that Glucose, BMI, and Age were important in understanding if a patient has diabetes. It was surprising seeing that pregnancy looked to be correlated in the initial heatmap, but turned out to be the least important feature in our final model. This could have been due to its high correlation with the Age feature.

## 2 Deep Learning

### 2.1 Neural Networks

NOTE: refer back to Perceptron Algorithm section (1.2) for basic building blocks.

**Error Function:** tells us the distance we are from the points, assigning more weight to the misclassified points and less to the correctly classified points. This should be continuous (not discrete).

**Continuous Predictions:** we now want to classify points based on probability since we are working in the continuous world. Using the *Sigmoid Function*, we classify all points with probability  $< 0.5$  as 0 and all points with probability  $> 0.5$  as 1.

**Sigmoid Function (Binary):** we now classify points as  $\hat{y} = \sigma(Wx + b)$ . The formula is  $\frac{1}{1+e^{-x}}$  and will give us a probability, which we classify based on a boundary line of 0.5.

**Softmax Function (Multi-class):** We apply the softmax function,  $\frac{e^{z_i}}{e^{z_1} + \dots + e^{z_n}}$ , to all of the linear function scores ( $Wx + b$ ) to find the probability for each class.

```
1 def softmax(L):
2     # Takes a list of numbers and returns the softmax values for each
3     p_class = []
4     for z in L:
5         p_class.append(np.exp(z)/np.sum(np.exp(L)))
6     return p_class
7
```

**One-Hot Encoding:** for labels, create a column for each label option and assign a 1 if it is the corresponding label, otherwise assign it a 0. In general, we will have  $n$  columns for  $n$  labels.